Analysis of Markov Chain Monte Carlo Algorithms for Bayesian Statistical Models

Long Zhao, Connor Kennedy, Shane Lubold,
Ethan Gwaltney, Patrick Dejesus, and Bethany Faz
Advisor: Dr. Jorge Carlos Román

Department of Mathematics and Statistics, San Diego State University

August 18, 2016

Contents

Abstract

For several decades Markov chain Monte Carlo (MCMC) methods have been used in many different fields. One of the most frequent areas in which they are used is Bayesian statistics where the user needs to approximate intractable posterior distributions and their associated integrals (e.g., posterior expectations). It is often possible to create a Markov chain that simulates approximate draws from a posterior distribution. Moreover, the simulated draws in the Markov chain can be used to easily construct MCMC estimators that converge to the unknown quantities of interest. In this manuscript we attempt to answer the following questions in the context of several widely used Bayesian statistical models.

(Q1) How long should the Markov chain be run before it is sufficiently close to the target distribution? In other words how much burn-in is necessary?

(Q2) How long should the Markov chain be run in order to make the root mean square error (RMSE) of estimation sufficiently small?

In the first part of this manuscript we provide an answer to (Q1). There is a theorem from Rosenthal (1995) that allows one to bound the total variation distance between a Markov chain (obeying certain conditions) and the target distribution. In previous research, the bounds resulting from the theorem have often been too large. They suggest that we need to run the chain for perhaps more than 10^{12} iterations which is impossible to do in a realistic context (and a in reasonable amount of time). We have worked with Rosenthal's theorem but we have approached proving the sufficient conditions for it in a different way that allow us to get much more reasonable bounds. The Markov chains that we studied were Gibbs samplers for 3 commonly used Bayesian statistical models: the one-sample normal model, the linear regression model, and the linear mixed model.

The second part of the manuscript contains a detailed analysis of the MCMC error of estimation and answers (Q2). A recent article Latuszyński et al. (2013) gives general bounds on the RMSE for Markov chains and functions of interest that satisfy certain convergence and integrability conditions. We were able to verify these conditions in the context of the Bayesian one-sample normal model and the linear regression model. This allowed us to calculate the minimum number of iterations needed to ensure that the RMSE is smaller than a user-specified threshold.

Finally, in the third part, the performances of the programming languages R, C++, JAGS, Matlab, and Julia for running MCMC algorithms are compared using a variety of Bayesian statistical models.

Part I

Markov Chain Convergence Analyses

Chapter 1

Introduction

In this chapter, we will introduce the basic concepts of our research using the 1-sample normal model as an example. We shall introduce the following Markov chain theory concepts: the Markov transition function, Harris ergodicity, and the total variation (TV) distance of two probability measures. We shall also demonstrate how to derive a widely used Markov chain Monte Carlo (MCMC) algorithm called the Gibbs sampler.

We shall first introduce several standard definitions. Given a discrete time Markov chain $\{\Phi^{(m)}\}_{m=0}^{\infty}$ with state space $\mathscr{X}$ that has a Borel σ-algebra $\mathscr{B}$, the Markov transition function is defined as

$$P(x, A) = \text{Pr}(\Phi^{(i+1)} \in A \mid \Phi^{(i)} = x)$$

where $x \in \mathscr{X}$ and $A \in \mathscr{B}$. For all of the Markov chains, we shall write the measure in terms of a conditional density, called the *Markov transition density* and denoted as $k(\cdot|x)$, where

$$P(x, A) = \int_A k(w|x)dw.$$

Throughout this manuscript we assume that the state space $\mathscr{X}$ is a subset of $\mathbb{R}^d$ for some d. Now we shall define the *m-step* Markov transition function. Let $P^m(x, A)$ be defined as

$$P^m(x, A) = \text{Pr}(\Phi^{(i+m)} \in A \mid \Phi^{(i)} = x).$$

We should point out that $P(x, A)$ can be regarded as the special case of $P^m(x, A)$ when $m = 1$.

In Bayesian statistics, the posterior distribution of parameters of interest is usually not available in closed form. Here we use a Markov chain, called a Gibbs sampler that gives us approximate samples from the posterior distribution. We shall demonstrate how to derive a Gibbs sampler in the following example.

Example 1. *Let $Y_1, Y_2, \dots, Y_n \overset{i.i.d}{\sim} N\left(\mu, \tau^{-1}\right)$, where τ denotes the precision parameter defined as the reciprocal of the variance. Suppose that the priors for μ and τ are independent and satisfy*

$$\mu \sim N\left(a, b^{-1}\right) \quad \perp \quad \tau \sim \textit{Gamma}(c, d),$$

where (a, b, c, d) are hyper-parameters. Let $\boldsymbol{y} = (y_1, y_2, \dots, y_n)^T$, and denote the sample mean by $\bar{y}$ and the sample standard deviation by s. Then, the posterior density is characterized by

$$f(\mu, \tau|\boldsymbol{y}) \propto \tau^{\frac{n}{2}+c-1} \cdot \textit{exp}\left\{-\frac{\tau}{2}\left[(n-1)s^2 + n(\mu-\bar{y})^2 + 2d\,\right] - \frac{b}{2}(\mu-a)^2\right\} \cdot I_{(0,\infty)}(\tau)\ .$$

Here $I_{(0,\infty)}(\tau)$ is the indicator function. To derive a Gibbs sampler, we shall first compute the conditional probability distributions. By completing the square, it easy to show that

$$\mu|\tau, \boldsymbol{y} \sim N\left(w\bar{y} + (1-w)a,\ \frac{1}{n\tau + b}\right),$$

$$\tau|\mu, \boldsymbol{y} \sim \textit{Gamma}\left(\frac{n}{2} + c,\ d + \frac{(n-1)s^2 + n(\mu-\bar{y})^2}{2}\right),$$

where $w = \frac{n\tau}{n\tau+b}$ can be regarded as the weight between the sample mean $\bar{y}$ and prior mean a. We shall construct the Gibbs Sampler that first updates τ and then μ. Then, if we denote the current state by (μ, τ) and the future state by (μ', τ'), the Markov chain follows the order of $(\mu', \tau') \to (\mu', \tau) \to (\mu, \tau)$. The state space $\mathscr{X} = \mathbb{R} \times \mathbb{R}^+$ and the Markov transition density for this Gibbs sampler is

$$k((\mu, \tau)|(\mu', \tau')) = f(\mu|\tau, \boldsymbol{y}) f(\tau|\mu', \boldsymbol{y}).$$

From now on, we shall suppress the dependence on $\boldsymbol{y}$, as the response vector is always fixed for our purpose. We will return to this example later.

We want to understand whether the Gibbs sampler converges to the posterior distribution. In other words, we want to know whether our "simulation" is valid or not. In order to discuss the convergence behavior, we need to define a collection of assumptions called *Harris ergodicity* which we shall denote by assumption $(\mathscr{H})$.

Definition 1. *We say that a Markov chain $\{\Phi^{(m)}\}_{m=0}^{\infty}$ satisfies assumption $(\mathscr{H})$ if*

1. *the chain has an invariant probability measure Π,*
2. *the chain is Π-irreducible,*
3. *the chain is aperiodic, and*
4. *the chain is Harris recurrent.*

We define the total variation distance as

$$\|P^m(x, \cdot) - \Pi(\cdot)\|_{TV} = \sup_{A \in \mathscr{B}} \|P^m(x, A) - \Pi(A)\|.$$

This is the greatest difference between the probability that after m steps the Markov chain lands in set A and the chance that a draw from the posterior distribution would be from set A. One can show that if a Markov chain satisfies $(\mathscr{H})$, then for every $x \in \mathscr{X}$

$$\|P^m(x, \cdot) - \Pi(\cdot)\|_{TV} \to 0 \quad \text{as } m \to \infty.$$

In other words, any Harris ergodic Markov chain eventually converges to the posterior distribution in total variation as we continue to run the Gibbs sampler. In practice, it is easy to show that a Markov chain satisfies $(\mathscr{H})$. For the type of Markov chain that we consider here, a sufficient condition for $(\mathscr{H})$ is that the Markov transition density is positive for (almost) all states. In all of the Gibbs samplers we consider $(\mathscr{H})$ is satisfied.

The remainder of this chapter is organized as follows. Section 1.1 is sub-divided into three parts: we shall define geometric ergodicity, the drift condition, and the minorization condition. As an example, we illustrate how to establish drift and minorization conditions for 1-sample normal model. In Section 1.2, we shall introduce Rosenthal's theorem which provides an upper bound for the total variation distance. The bound relies on the results from the drift and minorization conditions. To understand how well the bound performs, we shall present some results for the TV distance in the 1-sample normal model.

1.1 Geometric Ergodicity Via Drift and Minorization

Harris ergodicity does not indicate the rate at which a Markov chain approaches its invariant distribution. In order to know at what rate the chain is approaching the invariant distribution we will need to define another concept.

Definition 2. *A Markov chain $\{\Phi^{(m)}\}_{m=0}^{\infty}$ is geometrically ergodic if*

$$\|P^m(x, \cdot) - \Pi(\cdot)\|_{TV} \leq M(x) \upsilon^m$$

for all $x \in \mathscr{X}$ with some finite function $M(x)$ and constant $\upsilon \in (0, 1)$.

One approach to show that the chain converges at geometric rate is to establish a drift condition and an associated minorization condition (Rosenthal, 1995). We shall formally define these conditions in the next two subsections.

1.1.1 Drift Condition

We first provide a formal definition of drift condition:

Definition 3. *For a Markov chain $\{\Phi^{(m)}\}_{m=0}^{\infty}$, a drift condition holds if there exists some function $v : \mathscr{X} \to [0, \infty)$, some $0 \leqslant \rho < 1$, and some $L < \infty$ such that*

$$E[v(\Phi^{(m+1)})|\Phi^{(m)} = x] \le \rho v(x) + L \textit{ for all } x \in \mathscr{X}. \tag{1.1}$$

We shall provide a detailed example how to establish (1.1).

Example 1 (Continued). *We shall establish a drift condition using the function $v(\mu, \tau) = (\mu - \bar{y})^2$. Notice that function v does not depend on τ in its expression. For simplicity, we shall drop τ in our notation. By the law of iterated expectation,*

$$E\left[v(\mu^{(m+1)})|\, \mu^{(m)} = \mu\right] = E\left[E\left[v(\mu^{(m+1)})|\, \mu^{(m)} = \mu, \tau^{(m)} = \tau\right] \mid \mu^{(m)} = \mu\right] \tag{1.2}$$

$$= E\left[E\left[v(\mu^{(m+1)})|\, \tau^{(m)} = \tau\right] \mid \mu^{(m)} = \mu\right], \tag{1.3}$$

as $\mu_{m+1}|\tau_m$ is conditionally independent of μ_m. We shall first focus on the innermost expectation. Recall that $\mu|\tau, \boldsymbol{y} \sim N\left(w\bar{y} + (1-w)a,\ \frac{1}{n\tau+b}\right)$, so it is clear that $\mu - \bar{y}|\tau, \boldsymbol{y} \sim \left(w\bar{y} + (1-w)a - \bar{y},\ \frac{1}{n\tau+b}\right)$, where $w = \frac{n\tau}{n\tau+b}$. Then, the innermost expectation can be simplified into

$$\begin{aligned} E(v(\mu^{(m+1)})|\, \tau^{(m)} = \tau) &= E((\mu - \bar{y})^2|\tau^{(m)} = \tau) \\ &= Var(\mu|\tau, \boldsymbol{y}) + [E(\mu|\tau, \boldsymbol{y})]^2 \\ &= [w\bar{y} + (1-w)a - \bar{y}]^2 + \frac{1}{n\tau + b} \\ &= (1-w)^2 (a - \bar{y})^2 + \frac{1}{n\tau + b}\,. \end{aligned} \tag{1.4}$$

Before we compute the outer expectation, we shall obtain an upper bound for the innermost expectation in (1.3). In this example, we shall provide readers with three types of bounds.

Type 1 Bound:
It is clear that $(1-w)^2 = \left(\frac{b}{n\tau+b}\right)^2 \leqslant 1$, and $\frac{1}{n\tau+b} \leqslant \frac{1}{b}$. Then,

$$E(v(\mu^{(m+1)})|\tau^{(m)} = \tau) \leqslant (a - \bar{y})^2 + \frac{1}{b}\,.$$

In this way, we bound the innermost expectation by a constant. Then, when we apply the outer expectation, it is clear that

$$\begin{aligned} E\left[v(\mu^{(m+1)})|\mu^{(m)} = \mu\right] &\leqslant E\left[(a - \bar{y})^2 + \frac{1}{b} \mid \mu^{(m)} = \mu\right] \\ &= (a - \bar{y})^2 + \frac{1}{b} \\ &\leqslant \rho v(\mu) + (a - \bar{y})^2 + \frac{1}{b}, \end{aligned}$$

where ρ can be any constant $\in [0, 1)$ and $L = (a - y)^2 + b^{-1}$. We have established the drift condition for Type 1 bound.

Notice that the Type 1 bound does not require any additional conditions on the hyper-parameter (a, b, c, d) and $\boldsymbol{y}$. In addition, ρ can be set to be as small as zero; yet, L can be very large due to the effects of the hyper-parameter and the data set. As we shall see later, the Type 1 bound on the innermost expectation does not provide good bounds on the TV distance.

Type 2 Bound:
Alternatively, we can bound the innermost expectation by a function of τ. It is clear that

$$(1-w)^2 = \left(\frac{b}{n\tau + b}\right)^2 \leqslant 1, \ \textit{ and } \ \frac{1}{n\tau + b} \leqslant \frac{1}{n\tau}.$$

Therefore, we have

$$E(v(\mu^{(m+1)})|\tau^{(m)} = \tau) \leqslant (a-\bar{y})^2 + \frac{1}{n\tau}\ .$$

By formula (5), it is easy to show that

$$E(\tau^{-1}|\ \mu) = \frac{2d + (n-1)s^2 + n(\mu - \bar{y})^2}{n + 2c - 2},$$

provided that $n + 2c - 2 > 0$. Then, when we apply the outer expectation, it is clear that

$$\begin{aligned} E\left[v(\mu^{(m+1)})|\mu^{(m)} = \mu\right] &\leqslant E\left[(a-\bar{y})^2 + \frac{1}{n\tau}\ |\ \mu^{(m)} = \mu\right] \\ &= (a-\bar{y})^2 + \frac{2d + (n-1)s^2 + n(\mu-\bar{y})^2}{n^2 + 2cn - 2n} \\ &= \frac{1}{n+2c-2}v(\mu) + (a-\bar{y})^2 + \frac{2d + (n-1)s^2}{n^2+2cn-2n}. \end{aligned}$$

When $n + 2c > 3$, we have

$$\rho = \frac{1}{n+2c-2} \in [0,1) \ \ and \ \ L = (a-\bar{y})^2 + \frac{2d+(n-1)s^2}{n^2+2cn-2n} < \infty.$$

We have established the drift condition for Type 2 bound.

The Type 2 bound can control the size of L by the sample size n with the price of non-zero ρ and some conditions on (a, b, c, d) and $\boldsymbol{y}$. For any sample with decent sample size, L will be much smaller in the Type 2 bound than the Type 1 bound, consequently leading to a better TV distance. Notice that the Type 2 bound requires that $n + 2c > 3$, which is a very weak condition.

Type 3 Bound:
Finally, we present a different method to bound the innermost expectation as a function of τ. It is clear that

$$(1-w)^2 = \left(\frac{b}{n\tau+b}\right)^2 = \frac{b^2}{(n\tau)^2 + 2n\tau b + b^2} \leqslant \frac{b^2}{2n\tau b} = \frac{b}{2n\tau},$$

and we use the bound

$$\frac{1}{n\tau+b} \leqslant \frac{1}{n\tau}.$$

Then, we have

$$E(v(\mu^{(m+1)})|\tau^{(m)} = \tau) \leqslant \frac{b(a-\bar{y})^2}{2n\tau} + \frac{1}{n\tau} = \frac{b(a-\bar{y})^2+2}{2n\tau}\ .$$

Now, we can follow the same kind of computation in Type 2 bound. We eventually have that, if $n+2c > \frac{b(a-\bar{y})^2}{2}+3$,

$$E\left[v(\mu^{(m+1)})|\mu^{(m)} = \mu\right] \leqslant \frac{b(a-\bar{y})^2+2}{2(2c+n-2)}\ v(\mu) + \left[2 + b(a-\bar{y})^2\right]\frac{2d+(n-1)s^2}{4cn+2n^2-4n},$$

where

$$\rho = \frac{b(a-\bar{y})^2+2}{4c+2n-4} \in [0,1) \ \ and \ \ L = \left[2+b(a-\bar{y})^2\right]\frac{2d+(n-1)s^2}{4cn+2n^2-4n} < \infty.$$

We have established the drift condition for Type 3 bound.

Type 3 bound requires that $n + 2c > b(a-\bar{y})^2 + 3$, which is a stronger condition. As most priors give comparatively large variance, then b is relatively small, and a data set with decent sample size will satisfy the condition for Type 3 Bound. The advantage of Type 3 bound is that when the sample size is decent, it has a much smaller L than the Type 2 and the Type 1 bound. The property will significantly help when we establish the minorization condition, which we will introduce in the next subsection.

1.1.2 Minorization Condition

We shall now formally define a minorization condition.

Definition 4. *A minorization condition holds if there exists a probability measure Q on $\mathscr{B}$ and some set C for which $\pi(C) > 0$ such that*

$$P(x, A) \geqslant \epsilon\, Q(A) \quad \textit{for all } x \in C, A \in \mathscr{B}\ , \tag{1.5}$$

where ϵ is a real number in $(0, 1)$. The set C is called a small set.

Recall that a drift condition and an associated minorization condition for Φ is sufficient to verify that Φ is geometrically ergodic. More specifically, the chain is geometrically ergodic if it satisfies (1.1) and (1.5) with $C = \{x \in \mathscr{X} : v(x) \leqslant \delta\}$ and any δ larger than $2L/(1-\rho)$ Rosenthal (1995). We shall demonstrate how to establish a minorization in our toy example.

Example 1 (Continued)**.** *Let $C_{\mu,\tau} := \{(\mu', \tau') : (\mu' - \bar{y})^2 \leqslant \delta\}$, where $\delta > 0$. Suppose that we can find a density $q(\mu, \tau)$ on $\mathscr{X} = \mathbb{R} \times \mathbb{R}_+$ and an $\epsilon > 0$ such that whenever $\mu' \in C_{\mu,\tau}$,*

$$k((\mu,\tau)|(\mu',\tau')) = f(\mu|\tau)f(\tau|\mu') \geqslant \epsilon\, q(\mu,\tau) \quad \textit{for all } (\mu,\tau) \in \mathscr{X}. \tag{1.6}$$

Let $Q(\cdot)$ be the probability measure associated with the density q. Then for any set A and any $(\mu', \tau') \in C_{\mu,\tau}$, we have

$$\begin{aligned} P((\mu',\tau'),A) &= \int_A k((\mu,\tau)|(\mu',\tau'))d\mu d\tau \\ &\leqslant \epsilon \int_A q(\mu,\tau)d\mu d\tau = \epsilon Q(A), \end{aligned}$$

and hence (1.5) is established. We now construct a $q(\mu,\tau)$ and an $\epsilon > 0$ that satisfy (1.6).

Recall that $C_{\mu,\tau} := \{(\mu',\tau') : (\mu' - \bar{y})^2 \leqslant \delta\}$ and note that for any $(\mu',\tau') \in C_{\mu,\tau}$ we have

$$f(\mu,\tau)f(\tau,\mu') \geqslant f(\mu|\tau) \inf_{(\mu',\tau') \in C_{\mu,\tau}} f(\tau|\mu')$$

Recall that $\tau|\mu' \sim$ Gamma $\left(c + \frac{n}{2}, d + \frac{(n-1)s^2 + n(\mu'-\bar{y})^2}{2}\right)$. We can show that $g(\tau) := \inf_{(\mu',\tau') \in C_{\mu,\tau}} f(\tau|\mu')$ can be written in closed form. By Lemma (6) in the appendix, we have

$$\begin{aligned} g(\tau) &= \inf_{(\mu',\tau') \in C_{\mu,\tau}} G\left(c + \frac{n}{2}, d + \frac{(n-1)s^2 + n(\mu'-\bar{y})^2}{2}\right) \\ &= \begin{cases} G\left(c + \frac{n}{2}, d + \frac{(n-1)s^2}{2}\right), & \tau \leqslant \tau^* \\ G\left(c + \frac{n}{2}, d + \frac{(n-1)s^2 + n\delta}{2}\right), & \tau > \tau^* \end{cases} \end{aligned}$$

where

$$\tau^* = \frac{2c+n}{n\delta} \log\left(1 + \frac{n\delta}{2d + (n-1)s^2}\right).$$

Now put

$$\epsilon = \int_{\mathbb{R}} f(\mu|\tau)g(\tau)d\mu = \int_{\mathbb{R}_+} g(\tau)d\tau$$

The equality above can be shown by the application of Fubini's theorem. Then, the minorization condition is satisfied with this ϵ and the density $q(\mu,\tau) = \epsilon^{-1} f(\mu|\tau)g(\tau)$. Note that ϵ can be calculated with two evaluations of the incomplete gamma function. We will return to this model soon.

1.2 Rosenthal's Theorem for Total Variation Distance

Suppose that the Markov chain Φ satisfies assumption ($\mathscr{A}$). Here is a slightly simplified version of the Rosenthal (1995) result.

Theorem 1. *(Rosenthal, 1995) Suppose that Φ satisfies the drift condition and the minorization condition on $C = \{x : V(x) \leq \delta\}$ where δ is any number larger than $\frac{2L}{1-\rho}$. Let $\Phi_0 = x$ and define two constants as follows*

$$\eta = \frac{1+\delta}{1+2L+\delta\rho} \quad \textit{and} \quad U = 1 + 2(\rho\delta + L)$$

Then for any $0 < r < 1$

$$\|P^{(m)}(x,\cdot) - \Pi(\cdot)\|_{TV} \leq (1-\epsilon)^{rm} + \left(\frac{U^r}{\eta^{1-r}}\right)^m \left(1 + \frac{L}{1-\rho} + V(x)\right)$$

When applying this result, users have some freedom to choose the values of δ and r. In order to let the bound decrease as the number of iteration grows, users need to specify the values of δ and r such that $\frac{U^r}{\eta^{1-r}} < 1$. Furthermore, from our experience, slight changes in δ and r can potentially lead to wildly different results. We shall apply Theorem (1) in a realistic setting to exemplify how the bound works.

Example 1 (Continued)**.** *Suppose that our sample has size $n = 100$, mean $\bar{y} = 110$, and standard deviation $s = 13$. Recall the prior hyperparameters and choose $a = 120, b \approx 0.027, c \approx 21$, and $d \approx 2351$. Then the drift function established in Section (1.1.1) holds with the following $\rho's$ and $L's$ shown in the table. Notice that $\delta = 2L/(1-\rho) + \kappa$, where $\kappa > 0$ and we have full control of its size. We apply a two-dimensional optimization technique to find κ and r so that the total variation distance is less than 0.01 given the smallest number of iteration, m.*

Bound	ρ	L	κ	r	Expression	m
Type 1	0	136.97	31.98	0.0192	$(0.9999917)^m + 137.97(0.999963)^m$	554,000
Type 2	0.0071	101.66	31.60	0.0258	$(0.9999162)^m + 103.39(0.999697)^m$	55,000
Type 3	0.0168	3.90	20.03	0.2288	$(0.7874611)^m + 4.97(0.723405)^m$	22

Within this setting, we can clearly see the effects of the different types of bounds. Particularly, thanks to a much smaller L, the Type 3 bound performs incredibly better than the other two. Recall that Type 1 bound does not require any additional condition on the hyper-parameter (a, b, c, d) and $\boldsymbol{y}$, and ρ can be set to as small as 0. However, these properties do not give the Type 1 bound much advantage in our setting, as ρ in the other two bounds are very small as well. Type 2 bound performs better than Type 1, but its L still has a large magnitude as it contains the term $(a - \bar{y})^2$, which equals 100 in our setting.

The L term in Type 3 bound is significantly smaller because some portion of the term $(a - \bar{y})^2$ is moved to ρ and the remaining part in L is multiplied by $\frac{b(2d+(n-1)s^2)}{4cn+2n^2-4n}$, which is a very small value given a decent sample size, n. Admittedly, we do require a stronger condition on the hyper-parameter (a, b, c, d) and $\boldsymbol{y}$ so that ρ does not exceed 1. The condition states that $n + 2c > \frac{b(a-\bar{y})^2}{2} + 3$. Yet, notice that when the prior information about μ is not very precise, b is very likely to be smaller than 1. Given a decent sample size, the condition should be easily met in most circumstances.

Throughout this example, we demonstrate how to establish drift condition and minorization condition in the 1-sample Normal Model. By examining the total variation distance of the three different types of bound, we understand their performance in a practical way. The example illustrates that when working with more advanced models in further context, we should try to apply (and possibly, generalize) the Type 3 bound, which performs much better than the traditional Type 1 and Type 2 bounds.

Chapter 2

Linear Regression Model

We have so far demonstrated methods by which we can calculate tight bounds for the total variation distance in the 1-sample normal model's Gibbs sampler. We shall now turn our attention to the more complicated linear regression model. Several of the steps used to obtain tight upper bounds will be motivated by similar steps that were used in the 1-sample model. In Section 2.1, the model is introduced. In Section 2.2, several approaches to demonstrating a drift condition are presented. Next, in Section 2.3, the associated minorization condition is proven. Finally in Section 2.4 some numerical results for the TV distance bounds are presented using 2 models and several methods of proving the drift condition.

2.1 The Model and the Gibbs sampler

We recall that the Bayesian linear regression model is defined in the following way.

$$\tilde{Y}|\tilde{\beta},\sigma\sim N_n(X\tilde{\beta}, I_n\sigma^2)$$

$\tilde{Y}$ is a response data vector with dimensions $n \times 1$, X is a fixed nonzero data matrix with dimensions $n \times p$, $\tilde{\beta}$ is a random parameter vector with dimensions $p \times 1$ and σ is the random standard deviation. Within this model we assume the following independent proper priors on β and τ, where $\tau = 1/\sigma^2$.

$$\tilde{\beta} \sim N_p(\mu_\beta, \Sigma_\beta) \quad \perp \quad \tau \sim \text{Gamma}(a, b)$$

Each of the hyper parameters $\mu_\beta, \Sigma_\beta, a, b$ is constant and assumed to be within the correct range for these posteriors to be proper. Before we continue we will reparameterize the model to make some calculations easier. Define $\beta = \tilde{\beta} - \mu_\beta$ so that β has a prior mean $\vec{0}$. Also define $Y = \tilde{Y} - X\mu_\beta$. Note that we can write the model in the following way:

$$Y|\beta,\sigma\sim N_n(X\beta, I_n\sigma^2)$$

$$\beta \sim N_p(\vec{0}, \Sigma_\beta) \quad \perp \quad \tau \sim \text{Gamma}(a, b)$$

We shall work with the intractable posterior distribution for this reparameterized version of the model for the remainder of this section. We note by Lemma 7 in Appendix B that the Gibbs sampler for this model converges at the same rate as the Gibbs sampler for the original model. Now as before we cannot get the posteriors in closed form but one can easily derive the following conditional posteriors for τ and β. The conditional posterior for τ is given by:

$$\tau|\beta, Y, X \sim \text{Gamma}\left(a + \frac{n}{2}, b + \frac{\hat{\sigma}^2(n-p) + \left\|X(\beta - \hat{\beta})\right\|^2}{2}\right)$$

$$\hat{\sigma}^2 = \frac{\left\|Y - X\hat{\beta}\right\|^2}{n-p}, \quad \hat{\beta} = (X^TX)^+X^TY,$$

where $(X^TX)^+$ is the Moore Penrose inverse of X^TX. The conditional posterior for β is given by:

$$\beta|\tau, Y, X \sim N_p(\tau\Psi_\tau X^TY, \Psi_\tau), \qquad \Psi_\tau = [\tau X^TX + \Sigma_\beta^{-1}]^{-1}.$$

Note that similarly to the 1-sample case we shall be suppressing dependencies on Y and X in our notation from here on as we assume that they are fixed, known matrices. We will also write $Y = y$ as we consider Y to be

a random vector and y to be an observed data vector.
We shall be considering the Gibbs sampler $\{(\beta^{(m)}, \tau^{(m)})\}_{m=0}^{\infty}$ which updates τ then β such that if we start with an initial state (β', τ') and reach a final state (β, τ) it will update in two steps. First it updates τ' to τ by making a draw from τ's conditional gamma density given β' and then it updates β' to β by making a draw from $\beta' s$ conditional normal density given τ. Thus the update order is: $(\beta', \tau') \to (\beta', \tau) \to (\beta, \tau)$.

2.2 Drift Condition

As in Example 1, we will be considering three different types of bounds used to prove the drift condition for the linear regression model. Each of these bounds is similar to one from Example 1 although they are not exactly the same (when the linear model is specialized to the 1-sample model). The bounds that were presented earlier act mostly as a motivation for our overall strategy in bounding the expectation here.

2.2.1 Method 1 Bound

In our first method we want to see if we can prove the drift condition by bounding the entire expectation by a single constant. If we can do this we will have something similar to a Type 1 bound for the linear regression model. We don't expect the bound to perform very well but we want to see if it is at all usable.

Proposition 1. *There exist a finite constant $L = \|P_X y\|^2 + tr(X^T X \Sigma_\beta)$, where P_X is the projection matrix of X, such that, for every $\beta \in \mathbb{R}^p$,*

$$E\big(v(\beta^{(m+1)})|\beta^{(m)}\big) \leq L \ ,$$

where the drift function is defined as

$$v(\beta, \tau) = v(\beta) = \|X(\beta - \hat{\beta})\|^2 \ .$$

We see that this clearly implies that our Gibbs sampler obeys the drift condition as this is just the case where $\rho = 0$. We notice that the drift function here does not depend on the previous value of τ, but we know that the Gibbs sampler doesn't either so we believe this shouldn't cause an issue. Now to the proof:

Proof of Proposition. 1 We begin with the following by the law of iterated expectations.

$$E(\|X(\beta^{(m+1)} - \hat{\beta})\|^2|\beta^{(m)}) = E(E(\|X(\beta^{(m+1)} - \hat{\beta})\|^2|\tau^{(m+1)}, \beta^{(m)})|\beta^{(m)}) \ .$$

Note that if we can bound the inner expectation by a constant then taking the outer expectation will leave it unchanged. We shall focus on the inner expectation for the moment. Note that

$$E(\|X(\beta^{(m+1)} - \hat{\beta})\|^2|\tau^{(m+1)}, \beta^{(m)}) = E\left((\beta^{(m+1)} - \hat{\beta})^T X^T X(\beta^{(m+1)} - \hat{\beta})|\tau^{(m+1)} = \tau\right) \ .$$

This is a standard situation for the expectation of a normally distributed vector and is equal to the following:

$$E((\beta^{(m+1)} - \hat{\beta})^T X^T X(\beta^{(m+1)} - \hat{\beta})|\tau) = \text{tr}(X^T X \Psi_\tau) + (E(\beta^{(m+1)} - \hat{\beta}|\tau)^T X^T X E(\beta^{(m+1)} - \hat{\beta}|\tau) \ . \tag{2.1}$$

Recall that if A is a non-negative definite matrix then $\text{tr}(A) \geq 0$. If A and B are symmetric matrices (of the same dimension) such that $B - A$ is non-negative definite, we write $A \preceq B$. Also, if $A \preceq B$ then $\text{tr}(A) \leq \text{tr}(B)$. Furthermore, if A and B are positive definite matrices, then $A \preceq B$ if and only if $B^{-1} \preceq A^{-1}$. Since $\Sigma_\beta^{-1} \preceq \Psi_\tau^{-1}$, it follows that $\Psi_\tau \preceq \Sigma_\beta$. This implies that

$$\text{tr}(X^T X \Psi_\tau) = \text{tr}(X \Psi_\tau X^T) \leq \text{tr}(X \Sigma_\beta X^T) = \text{tr}(X^T X \Sigma_\beta) \ , \tag{2.2}$$

since $X \Psi_\tau X^T \preceq X \Sigma_\beta X^T$.

We now focus on the last term in (2.1),

$$(E(\beta^{(m+1)} - \hat{\beta})|\tau)^T X^T X E(\beta^{(m+1)} - \hat{\beta}|\tau) = \|X[\tau \Psi_\tau X^T y - \hat{\beta}]\|^2 \ . \tag{2.3}$$

For this term we may make use of the following Lemma which is proven in Appendix B:

Lemma 1. *Define $g(\tau) = \|X(\tau \Psi_\tau X^T y - \hat{\beta})\|^2$. Then $g(\tau)$ is monotone nonincreasing and convex.*

By Lemma 1 we see that the supremum of $g(\tau)$ must be $g(0)$ which is well defined. It is then possible to obtain the following bound:

$$\|X(\tau\Psi_\tau X^T y - \hat{\beta})\|^2 \leq \sup_\tau \|X(\tau\Psi_\tau X^T y - \hat{\beta})\|^2 = \|P_X y\|^2 \ . \tag{2.4}$$

By combining (2.2) and (2.4) we may conclude.

$$E(\|X(\beta^{(m+1)} - \hat{\beta})\|^2|\tau, \beta^{(m)}) \leq \|P_X y\|^2 + tr(X^T X \Sigma_\beta) \ .$$

We see that this is a constant which will be unchanged upon applying the outer expectation. Thus we may make the following overall conclusion

$$E(\|X(\beta^{(m+1)} - \hat{\beta})\|^2|\beta^{(m)}) \leq \|P_X y\|^2 + tr(X^T X \Sigma_\beta) \ . \tag{2.5}$$

We see that this is precisely the inequality presented in proposition 1 which completes our proof. □

This proof is sufficient to show that the Gibbs sampler obeys the drift condition but as we shall see in Section 2.4, the results for the TV distance with this bound are quite poor. We may improve upon this method using several alternate approaches.

2.2.2 Method 2 Bound

Our goal here is to find something comparable to the type 2 bound. The trace term comes from the variance of a random vector similar to the $\frac{1}{n\tau+b}$ term from the variance of a random variable in the 1-sample normal model. We will look for some sort of inverse τ function as an upper bound of the trace.

Proposition 2. *If X has full rank, $n + 2a > p + 2$, and $\frac{n}{2} + a > 1$ then there exists*

$$\rho = \frac{p}{n + 2a - 2} < 1 \quad \textit{and} \quad L = \|P_X y\|^2 + p\left(\frac{2b + (n-p)\hat{\sigma}^2}{n + 2a - 2}\right)$$

such that, for every $\beta \in \mathbb{R}^p$,

$$E\big(v(\beta^{(m+1)})|\beta^{(m)}\big) \leq \rho\, v(\beta^{(m)}) + L \ ,$$

We see that ρ and L are constants with the necessary properties for this to imply the drift condition holds. Note that while 3 assumptions are needed for this proof, none of them is very strong so our proof is still quite general.

Proof of Proposition 2. Using this method we obtain the same bound that we did in (2.4) but a different bound for the trace term of (2.1). If we assume that X has full rank then $(X^T X)^{-1}$ is defined thus we may say that $\Psi_\tau \preceq (\tau X^T X)^{-1}$. This allows us to obtain the alternate bound

$$\text{tr}(X^T X \Psi_\tau) = \text{tr}(X \Psi_\tau X^T) \leq \text{tr}(X(\tau X^T X)^{-1} X^T) = \text{tr}(X^T X (\tau X^T X)^{-1}) = \frac{p}{\tau} \ .$$

The expectation for $\frac{1}{\tau}$ where τ is Gamma distributed is given by Lemma 5 in Appendix A and is well defined so long as $\frac{n}{2} + a > 1$. Applying the expectation we get the following,

$$E\left(\frac{p}{\tau}|\beta^{(m)}\right) = p\left(\frac{2b + (n-p)\hat{\sigma}^2}{n + 2a - 2}\right) + \left(\frac{p}{n + 2a - 2}\right)\|X(\beta^{(m)} - \hat{\beta})\|^2 \ . \tag{2.6}$$

If this result is combined with (2.4) then the following overall inequality is true.

$$E(\|X(\beta^{(m+1)} - \hat{\beta})\|^2|\beta^{(m)}) \leq \left(\frac{p}{n + 2a - 2}\right) v(\beta^{(m)}) + \|P_X y\|^2 + p\left(\frac{2b + (n-p)\hat{\sigma}^2}{n + 2a - 2}\right) \ . \tag{2.7}$$

We see this is precisely the form of proposition 2 thus the proof is complete. □

2.2.3 Method 3 Bound

Here our goal is to find one more method by which we can prove the drift condition which gives results comparable to the Type 3 bound. We first present some motivation for the method we employ. We want to find a function of τ that bounds (2.3) rather than bounding it by a constant. If we can find a function with similar properties as an upper bound this should give us something much tighter. We would also like to bound the trace term from (2.1) by a close function without having to assume that X has full rank. Ideally we want to find a function that bounds both simultaneously. For this purpose we use the following lemma which is proven in Appendix B holds true:

Lemma 2. *Define $h(\tau) := tr(X^T X \Psi_\tau)$. Then $h(\tau)$ is monotone nonincreasing and convex.*

We see that both $g(\tau)$ and $h(\tau)$ are nonnegative, monotone nonincreasing, and convex thus their sum, $G(\tau) := g(\tau) + h(\tau)$ must retain these properties as well. A function of the form $\frac{\gamma_1}{\tau} + \gamma_2$, where each γ_i is a positive constant, also has these properties. Its expectation returns our drift function in a simple way as well. If we can find a function of this form which acts as an upper bound of $G(\tau)$ then it should act as a tight upper bound. With that in mind we present the following proposition.

Proposition 3. *If $\frac{n}{2} + a > 1$ then there exist positive constants γ_1, γ_2 such that if*

$$\rho = \frac{\gamma_1}{n+2a-2} < 1 \qquad L = \gamma_1 \left(\frac{2b + (n-p)\hat{\sigma}^2}{n+2a-2} \right) + \gamma_2 \; .$$

then for every $\beta \in \mathbb{R}^p$,

$$E\big(v(\beta^{(m+1)})|\beta^{(m)}\big) \leq \rho \, v(\beta^{(m)}) + L \; .$$

Proof of Proposition 3. We may calculate the values for γ_1 and γ_2 in the following way. Let $\iota \in \mathbb{R}_+$ be a constant. Define $C_\iota := (0, \iota]$. Then γ_1, γ_2 are calculated as

$$\gamma_1 = \sup_{C_\iota}(\tau(G(\tau)) \quad , \quad \gamma_2 = G(\iota).$$

We must resort to numerical methods to calculate γ_1 but so long as ι is not too large we get accurate results. We note that by the definitions of γ_1 and γ_2 the following inequalities must hold.

$$G(\tau) \leq \frac{\gamma_1}{\tau}, \text{ if } \tau \in C_\iota \quad \text{and} \quad G(\tau) \leq \gamma_2, \text{ if } \tau \notin C_\iota \; .$$

The inequality for γ_1 follows directly from its definition and the inequality for γ_2 follows from the fact that $G(\tau)$ is monotone nonincreasing. If we take the sum of these functions we get

$$G(\tau) \leq \frac{\gamma_1}{\tau} + \gamma_2 \text{ for all } \tau \in \mathbb{R}_+.$$

If we then apply the outer expectation to this function we will get the following final bound on $E\big(v(\beta^{(m+1)})|\beta^{(m)}\big)$

$$E\big(v(\beta^{(m+1)})|\beta^{(m)}\big) \leq \frac{\gamma_1}{n+2a-2} v(\beta^{(m)}) + \gamma_1 \left(\frac{2b + (n-p)\hat{\sigma}^2}{n+2a-2} \right) + \gamma_2 \; . \tag{2.8}$$

We see that this is exactly the inequality from proposition 3 thus so long as $\gamma_1 < n + 2a - 2$ then $\rho < 1$ and we have our proof.

We may obtain γ_1 such that $\gamma_1 < n + 2a - 2$ by the following method. We know that if $\tau = 0$ then $\tau(G(\tau)) = 0$. We also know that $\tau(G(\tau))$ is a continuous function by sum and product of continuous functions. This makes it possible to select ι such that γ_1 is as close to zero as we like. We thus have $\rho < 1$ and have completed our proof. □

We note in general that decreasing ι must either decrease the value of γ_1 or not change it as we are taking the supremum over a smaller set. Decreasing ι will however increase, or not change, the value of γ_2 due to $G(\tau)$ being a nonincreasing function. There is thus a tradeoff between the size of γ_1 and the size of γ_2 when we choose a value for ι.
It is interesting to see that both of the other methods for proving the drift condition just look like special cases of this one. If we allow $\iota = 0$ then $\gamma_1 = 0$ and $\gamma_2 = G(0)$ which is exactly the same value that was obtained for L in method 1. In method 2, we effectively have $\gamma_1 = p$ and $\gamma_2 = \|P_X y\|^2$. This shows that this method is much more flexible than the other two and why we are potentially able to obtain better bounds on the TV distance.

2.3 Minorization Condition

Now each of the methods in the previous section proves that the Gibbs sampler obeys a drift condition but in order to apply Rosenthal's theorem we must also prove an associated minorization condition. Conveniently we only need to consider a single minorization condition that works for each method.

Proposition 4. *The Markov chain $\{(\beta^{(m)}, \tau^{(m)})\}_{m=0}^{\infty}$ obeys a minorization condition.*

Proof of Proposition 4. The Markov transition density for $P((\beta, \tau), A)$ is defined as

$$k(\beta, \tau|\beta', \tau') = f(\beta|\tau)f(\tau|\beta').$$

We shall be bounding this density below by another density function which does not depend on the previous step to establish the minorization condition. For our set C we define $C_\beta := \{(\beta', \tau') : \|X(\beta' - \hat{\beta})\|^2 \leq \delta\}$ where $\delta \geq \frac{2L}{1-\rho}$ is a constant. We see that the set does not restrict τ' but this is to be expected as we know that (β, τ) does not depend on τ'. Note that for any $\beta' \in C_\beta$ we have

$$f(\beta|\tau)f(\tau|\beta') \geqslant f(\beta|\tau) \inf_{\beta' \in C_\beta} f(\tau|\beta')$$

Recall that:

$$\tau|\beta' \sim \text{Gamma}\left(a + \frac{n}{2}, b + \frac{(n-p)\hat{\sigma}^2 + \|X(\beta' - \hat{\beta})\|^2}{2}\right), \tag{2.9}$$

with

$$\hat{\sigma} = \frac{\|Y - X\hat{\beta}\|}{\sqrt{n-p}} \quad \hat{\beta} = (X^T X)^{-1} X^T Y.$$

$w(\tau) := \inf_{\beta' \in C_\beta} f(\tau|\beta')$ can be written in closed form by the following proposition.

Proposition 5. *The function $w(\tau)$ has the following form which is proven in Appendix B.*

$$w(\tau) = \inf_{\beta' \in C_\beta} G\left(a + \frac{n}{2}, b + \frac{(n-p)\hat{\sigma}^2 + \|X(\beta' - \hat{\beta})\|^2}{2}\right)$$

$$= \begin{cases} G\left(a + \frac{n}{2}, b + \frac{(n-p)\hat{\sigma}^2}{2}\right) & \tau' \leqslant \tau^* \\ G\left(a + \frac{n}{2}, b + \frac{(n-p)\hat{\sigma}^2 + \delta}{2}\right) & \tau' > \tau^* \end{cases}$$

$$\tau^* = \frac{2a+n}{\delta} log\left(1 + \frac{\delta}{2b + (n-p)\hat{\sigma}^2}\right).$$

We see that this form follows from Lemma 6 which is proven in Appendix B similarly to the minorization for the 1-sample normal model. With a closed form for $w(\tau)$ we are now able to calculate ϵ:

$$\epsilon = \int_{\mathbb{R}} \int_{\mathbb{R}_+} f(\beta|\tau) w(\tau) d\tau d\beta = \int_{\mathbb{R}_+} w(\tau) d\tau .$$

Once again an application of Fubini's theorem is necessary. The final calculation may be done through two computations of the incomplete gamma distribution. Thus, the minorization condition is satisfied with this ϵ and the distribution whose density is $q(\beta, \tau) = \epsilon^{-1} f(\beta|\tau) w(\tau)$. □

2.4 Example: Bounds on the Total Variation Distance

We see that each of the methods we have used to prove the drift condition may be used to calculate TV bounds by Rosenthal's theorem. Each of them will be used with two concrete examples to see how they compare. We shall also consider one new method, denoted method 4, where we bound only the norm term from (2.3) via the numerical method and we bound the trace term separately using the full rank assumption. Each of these methods was used for two linear regression models on NBA 2015 data. In each case the values of r, κ have been optimized to give the best possible results. In all cases the value ι was set to be 10^5 although in general it would be possible to optimize ι and potentially get better results. We begin with the following model:

$$\log(\text{PPG})_i|\beta,\tau \sim \text{N}\left(\beta_1 + \beta_2\text{MIN}_i, \frac{1}{\tau}\right),$$

where PPG means points per game and MIN means average minutes on court. We note that in this model we used a sample size $n = 486$ and we removed players whose PPG where zero to ensure log(PPG) is well defined. We now present the following results on the TV bounds:

NBA Log(PPG) Model

Method	ρ	L	κ	r	m	Bound	Formula
1	0	72.89	-	-	-	1	-
2	.003919	1.600	3.068	.2067	55	.009151	$.9165^m + (.8762^m)2.606$
3	.003919	.1959	2.362	.3932	13	.006546	$.6484^m + (.6303^m)1.197$
4	.003944	.1972	2.362	.3926	13	.006621	$.6490^m + (.6307^m)1.198$

In method 1 the $(1-\epsilon)$ term in Rosenthal's theorem is unusable. This is because the value of ϵ is so small that when our algorithm calculates $1-\epsilon$ it is rounded to 1. The actual computed value of epsilon is extremely small so the number of iterations necessary to get useful bounds would be impossible to run. Unlike the Type 1 bound in the 1-sample normal model, method 1 here is not just the worst, it is completely unusuable. Each of the other methods however gives us useful results although methods 3 and 4 vastly outperform method 2. It is important to also note that method 4 performs just slightly worse than method 3 thus it seems that choosing to bound the trace term numerically or using the bound available when X has full rank makes little difference. We now present the second model that we considered:

$$\text{PF}_i|\beta,\tau \sim \text{N}\left(\beta_1 + \beta_2\text{STL}_i + \beta_3\text{BLK}_i + \beta_4\text{MIN}_i, \frac{1}{\tau}\right),$$

where PF means personal fouls, STL means steals, BLK means blocks, and MIN means average minutes on court. Each of these is measured per game. Here the sample size was $n = 492$. In this model the magnitude of the data was significantly higher than in the previous model and we believed this would cause the bounds to be higher. We now present the following bounds on the TV distance.

NBA PF Model

Method	ρ	L	κ	r	m	Bound	Formula
1	0	7815	-	-	-	1	-
2	.007978	2479	-	-	-	1	-
3	.0348	4.073	7.0261	.1502	81	.009643	$.9428^m + (.9014^m)5.219$
4	.0358	4.1958	7.075	.1469	85	.009802	$.9457^m + (.9054^m)5.352$

In this case both methods 1 and 2 give us a result where ϵ is too small for the $(1-\epsilon)$ term to approach zero in a reasonable amount of time. It seems that the bounds from Rosenthal's theorem become worse if the magnitude of the data becomes large. In the log(PPG) model method 2 was feasible for calculating useful bounds but that is no longer the case here. This gives us very strong motivation for using the method 3 bound. We do note that even with this model methods 3 and 4 are still quite close to each other. This strengthens our hypothesis that the full rank bound for the trace term is very close to the numerically derived bound.

We see that in the linear regression model we were able to obtain a bound that was similar to the Type 3 bound from the 1-sample model. As expected this bound vastly outperformed the first two bounds we developed. We hope to apply some of these ideas to Gibbs Samplers for more complicated models such as the linear mixed model.

Chapter 3

Linear Mixed Model

The general linear mixed model has a wide range of applications. Bayesian versions of this model require us to specify a prior distribution for the parameters but, unfortunately, any non-trivial prior leads to an intractable posterior density. In this chapter, revisit the work of Román and Hobert (2015) study the convergence properties of a (block) Gibbs sampler Markov chain based on a (conditionally) conjugate prior. In Section 3.1, we introduce the setup of the model and the construction of the Gibbs sampler. More detailed introduction can be found in Román and Hobert (2015). In Section 3.2, we define the drift function and establish the drift condition with a "Type 2" bound. In Section 3.3, we show that the Markov chain satisfies the minorization condition with a drift function similar to the one from the previous section. In Section 3.4, we apply Theorem 1 to understand the performance of our bound given a concrete setting. We end this chapter with a brief discussion of some future work that will improve the bound.

3.1 The Model and the Gibbs Sampler

The first stage of the Bayesian hierarchical model is

$$Y|\beta, u, \lambda \sim \mathrm{N}_N\left(X\beta + \sum_{i=1}^{r} Z_i u_i,\ \lambda_e^{-1} I\right),$$

where Y is an $N \times 1$ response vector, X and Z_i are known matrices of dimensions $N \times p$ and $N \times q_i$, respectively, β is a $p \times 1$ regression coefficient, u_i is a $q_i \times 1$ vector that represents the ith random factor in the model, $u := (u_1^T\ u_2^T \dots u_r^T)^T$, λ_e is the precision parameter associated with β, each λ_{u_i} is the precision parameter associated with u_i, and $\lambda := (\lambda_e\ \lambda_{u_1} \cdots \lambda_{u_r})^T$. Given λ, the random elements β and u are assumed to be mutually independent and the second stage specifies their prior distribution:

$$\beta|\lambda \sim \mathrm{N}(\mu_\beta, \Sigma_\beta) \quad \perp \quad u|\lambda \sim \mathrm{N}_q(0, \Lambda^{-1}),$$

where $\Lambda_u = \oplus_{i=1}^{r} \lambda_{u_i} I_{q_i}$ and $q = q_1 + q_2 + \cdots + q_r$. Finally, the third stage of the model specifies the distribution of the precision parameters, which are independent with marginals given by

$$\lambda_e \sim \text{Gamma}(a_e, b_e) \quad \perp \quad \lambda_{u_i} \sim \text{Gamma}(a_i, b_i), \text{ for } i = 1, 2, \dots, r.$$

The hyper-parameters $\mu_\beta, \Sigma_\beta,\ a = (a_e\ a_1 \dots a_r)^T$ and $b = (b_e\ b_1 \dots b_r)^T$ are all assumed to be known and are restricted to their usual ranges to ensure a proper prior.

To construct block Gibbs sampler, we shall first define $\theta = (\beta^T u^T)^T$, $Z = (Z_1\ Z_2 \dots Z_r)$, and $W = (X\ Z)$, so that

$$W\theta = X\beta + Zu = X\beta + \sum_{i=1}^{r} Z_i u_i.$$

Also, let $\boldsymbol{y}$ denote the observed response vector. One can show that

$$\lambda_e|\theta, \boldsymbol{y} \sim \text{Gamma}\left(a_e + \frac{N}{2},\ b_e + \frac{\|\boldsymbol{y} - W\theta\|^2}{2}\right),$$

and, for $i \in \{1, 2, \dots, r\}$,

$$\lambda_{u_i}|\theta, \boldsymbol{y} \sim \text{Gamma}\left(a_i + \frac{q_i}{2},\ b_i + \frac{\|u_i\|^2}{2}\right).$$

Then, we can construct a Markov chain $\{(\lambda_m, \theta_m)\}_{m=0}^{\infty}$ that lives on $\mathscr{X} = \mathbb{R}_+^{r+1} \times \mathbb{R}^{p+q}$. If the current state of the chain is (λ_m, θ_m), then the next state, $(\lambda_{m+1}, \theta_{m+1})$, is simulated in two steps. First, we draw λ_{m+1} from the conditional posterior density of λ given $\theta = \theta_m$, which is a product of $r + 1$ univariate gamma densities. Then, we draw θ_{m+1} from the conditional posterior density of θ given $\lambda = \lambda_{m+1}$, which is a $(p + q)$-dimensional multivariate normal density. In other words, the chain goes in the following order:

$$(\lambda_m, \theta_m) \to (\lambda_{m+1}, \theta_m) \to (\lambda_{m+1}, \theta_{m+1}).$$

It is clear that the two marginal sequences, $\{\lambda_m\}_{m=0}^{\infty}$ and $\{\theta_m\}_{m=0}^{\infty}$, are also Markov chains and their invariant densities are the marginal posterior distribution of λ and θ, respectively. One can show that all the three Markov chains satisfy assumption $(\mathscr{A})$, and geometric ergodicity is a solidarity property for these chains, Román (2012), Diaconis et al. (2008), and Roberts and Rosenthal (2001). We now state the main result in Román and Hobert (2015).

Theorem 2 (Román and Hobert (2015))**.** *The block Gibbs Markov chain, $\{\lambda_n, \theta_n)\}_{n=0}^{\infty}$, is geometrically ergodic if*

1. *X has full column rank,*
2. *$a_e > \frac{1}{2}(rank(Z) - N + 2)$, and*
3. *$min\{a_1 + \frac{q_1}{2}, \ldots, a_r + \frac{q_r}{2}\} > \frac{1}{2}(q - rank(Z)) + 1$.*

However, the proof of Theorem 2 does not directly lead to a workable bound for the total variation distance. To obtain a tight bound, we shall consider a slightly different drift function.

3.2 Drift Condition

Consider the following drift function for the Gibbs Markov chain,

$$v(\theta, \lambda) = v(\theta) = \alpha \|W(\theta - \hat{\theta})\|^2 + \|u\|^2,$$

where α is a positive constant and $\hat{\theta} = (W^T W)^+ W^T y$ where A^+ denotes the Moore-Penrose inverse of matrix A. Notice that

$$W\hat{\theta} = W(W^T W)^+ W^T y = P_W y,$$

where P_W denotes the projection matrix onto the column space of W. We shall compare our drift function $v(\theta)$ with the drift function $v'(\theta)$ used in the proof of Theorem 2, where

$$v'(\theta) = \alpha \|\boldsymbol{y} - W\theta\|^2 + \|u\|^2 \ .$$

By the orthogonal decomposition of $\boldsymbol{y}$ into $P_W \boldsymbol{y} + (I - P_W)\boldsymbol{y}$, it is easy to show that

$$v'(\theta) = \|(I - P_W)\boldsymbol{y}\|^2 + \|P_W \boldsymbol{y} - W\theta\|^2 = \|(I - P_W)\boldsymbol{y}\|^2 + v(\theta) \ ,$$

as W and y are known. Note that $\|(I - P_W)\boldsymbol{y}\|^2$ is a fixed constant. Since the two drift functions are off by a constant, it is clear that we can follow some ideas in the proof of Theorem 2 to establish the drift condition for the new function. In order to have a tighter bound, we shall assume (slightly) stronger assumptions than Theorem 2.

Proposition 6. *Suppose that X and Z have full column rank, and $a_e > \frac{1}{2}(2 + p + q - N)$. Then, there exists a positive constant α, a constant $\rho \in [0, 1)$, and a finite constant L such that for every $\theta' \in \mathbb{R}^{p+q}$,*

$$E(v(\theta)|\theta') \leqslant \rho v(\theta') + L,$$

where the drift function is defined as

$$v(\theta) = \alpha \|W(\theta - \hat{\theta})\|^2 + \|u\|^2.$$

Remark 1. *Notice that this proposition does not assume any conditions on $a_i + \frac{q_i}{2}$ for any $i \in \{1, 2, \ldots, r\}$, whereas the third statements in Theorem 2 requires some (fairly weak) assumptions.*

Remark 2. *Recall that in the linear regression model, we discussed how to reparameterize the model and set the prior mean of β to be zero. Here we shall follow the same procedure. It is clear that we can set $\mu_\beta = 0$ and simplify calculation without loss of generality.*

Remark 3. *For this remark we will need to introduce some new notation. Define $T_\lambda^{-1} = (\lambda_e X^T X + \Sigma_\beta^{-1})^{-1}$ and $Q_\lambda^{-1} = (\lambda_e Z^T M_\lambda Z + \Lambda_u)^{-1}$, where $M_\lambda = I - \lambda_e X T_\lambda^{-1} X^T$. These definitions follow from Román and Hobert (2015). From the proof of Theorem 2 in Román and Hobert (2015), we know that the main focus is to obtain upper bounds for the terms*

$$E(\|W(\theta-\hat{\theta})\|^2|\lambda) = tr(W Var(\theta|\lambda) W^T) + \|W(E(\theta|\lambda)-\hat{\theta})\|^2 \tag{3.1}$$

and

$$E(\|u\|^2|\lambda) = tr(Q_\lambda^{-1}) + \|E(u|\lambda)\|^2, \tag{3.2}$$

which are complicated functions of λ. This comes from the fact that

$$E(v(\theta)|\theta') = E(E(v(\theta)|\theta'\lambda)|\theta') \ , \tag{3.3}$$

and the inner expectation from (3.3) is equal to the sum of (3.1) and (3.2). Here the definitions of $Var(\theta|\lambda)$, $E(u|\lambda)$, and $E(\theta|\lambda)$ are:

$$E(\theta|\lambda, y) = \begin{bmatrix} T_\lambda^{-1}\lambda_e X^T y - \lambda_e^2 T_\lambda^{-1} X^T Z Q_\lambda^{-1} Z^T M_\lambda y \\ \lambda_e Q_\lambda^{-1} Z^T M_\lambda y \end{bmatrix},$$

$$Var(\theta|\lambda, y) = \begin{bmatrix} T_\lambda^{-1} + \lambda_e^2 T_\lambda^{-1} X^T Z Q_\lambda^{-1} Z^T X T_\lambda^{-1} & -\lambda_e T_\lambda^{-1} X^T Z Q_\lambda^{-1} \\ -\lambda_e Q_\lambda^{-1} Z^T X T_\lambda^{-1} & Q_\lambda^{-1} \end{bmatrix},$$

$$E(u|\lambda) = \lambda_e Q_\lambda^{-1} Z^T M_\lambda y \ .$$

Our goal in this section is to obtain a “Type 2” bound for the terms in (3.1) and (3.2), i.e. we want to bound the “trace” terms by some functions of λ, and we want to bound each “norm” term by its supremum.

Román and Hobert (2015) show that

$$\text{tr}(W\text{Var}(\theta|\lambda)W^T) = \text{tr}(ZQ_\lambda^{-1}Z^T) + \text{tr}(XT_\lambda^{-1}X^T) - \text{tr}((I-M_\lambda)ZQ_\lambda^{-1}Z^T(I+M_\lambda)). \tag{3.4}$$

We shall state some preliminary results that help to bound the “trace” term in (3.4). The proof of this first lemma is in Appendix C.

Lemma 3. *If rank$(X) = p$, then for all $\lambda \in \mathbb{R}_+^{r+1}$,*

1. *$tr(W Var(\theta|\lambda) W^T) = tr(X T_\lambda^{-1} X^T) + tr(M_\lambda Z Q_\lambda^{-1} Z^T M_\lambda)$,*
2. *$tr(W Var(\theta|\lambda) W^T) \le (p + rank(Z))\lambda_e^{-1}$.*

For the “trace” term in (3.2), we shall construct an upper bound of the form of $\frac{\gamma_1}{\lambda_e} + \gamma_2$ for some positive γ_1 and γ_2. To do so, we need the following lemma which is proven in Appendix D.

Lemma 4. *Let Q_λ be defined as before. Then for all $(\lambda_e, \lambda_{u_1}, \ldots, \lambda_{u_r}) \in \mathbb{R}_+^{r+1}$,*

$$\frac{\partial\, tr(Q_\lambda^{-1})}{\partial \lambda_e} < 0 \quad and \quad \frac{\partial\, tr(Q_\lambda^{-1})}{\partial \lambda_{u_i}} < 0.$$

We shall state the construction of the upper bound formally as a proposition.

Proposition 7. *Suppose that Z has full column rank. Define $h(\lambda_e) := tr\,[(\lambda_e Z^T M_\lambda Z)^{-1}]$. Let ι be some positive constant and C_ι be the set $(0, \iota]$. Define $\gamma_1 = \sup_{\lambda_e \in C_\iota}(\lambda_e \cdot h(\lambda_e))$, and $\gamma_2 = h(\iota)$. Then, γ_1 is well-defined, i.e. γ_1 is a finite constant; and,*

$$tr(Q_\lambda^{-1}) < \frac{\gamma_1}{\lambda_e} + \gamma_2 \quad \textit{for all } (\lambda_e, \lambda_{u_1}, \ldots, \lambda_{u_r}) \in \mathbb{R}_+^{r+1}.$$

Proof of Proposition 7. It is clear that

$$\lambda_e \cdot h(\lambda_e) = \lambda_e \cdot \text{tr}[\ (\lambda_e Z^T M_\lambda Z)^{-1}] = q\ \text{tr}[\ (Z^T M_\lambda Z)^{-1}\ ]\ .$$

Recall that M_λ is a positive-definite matrix, and tr(M_λ) is finite for all $\lambda_e \in C_\iota$. Then, it is easy to check that γ_1 is well-defined, i.e. γ_1 is a finite constant. Then, we shall construct an upper bound for $h(\lambda_e)$. Recall that

$$Q_\lambda = \lambda_e Z^T M_\lambda Z + \Lambda_u.$$

By Lemma 4, we know that for any fixed $\lambda_e \in \mathbb{R}_+$, tr$(Q_\lambda^{-1})$ is monotone decreasing with respect to each λ_{u_i}. Then, to obtain an upper bound for tr(Q_λ^{-1}), we shall set each $\lambda_{u_i} = 0$. Since Z has full column rank, we have

$$\text{tr}(Q_\lambda^{-1}) < \text{tr}[\,(\lambda_e Z^T M_\lambda Z)^{-1}] = h(\lambda_e).$$

By the results from Lemma 4, it is easy to show that $h(\lambda_e)$ is a monotone decreasing function. For any $\lambda_e \in C_\iota$, it is clear that

$$h(\lambda_e) \leqslant \frac{\gamma_1}{\lambda_e} < \frac{\gamma_1}{\lambda_e} + \gamma_2,$$

as $\gamma_1 = \sup_{\lambda_e \in C_\iota}(\lambda_e \cdot h(\lambda_e))$. For any $\lambda_e \in (\iota, \infty)$, it is clear that

$$h(\lambda_e) < \gamma_2 < \frac{\gamma_1}{\lambda_e} + \gamma_2,$$

as $h(\lambda_e)$ is a monotone decreasing function. Hence, we conclude that

$$\text{tr}(Q_\lambda^{-1}) < h(\lambda_e) < \frac{\gamma_1}{\lambda_e} + \gamma_2.$$

$\square$

Román and Hobert (2015) show that $\|E(u|\lambda)\|$ and $\|\boldsymbol{y} - WE(\theta|\lambda)\|$ are bounded above by some constants. Then, it is easy check $\|W(E(\theta|\lambda) - \hat{\theta})\|$ is also bounded above by a constant, as

$$\|W(E(\theta|\lambda) - \hat{\theta})\|^2 = \|\boldsymbol{y} - WE(\theta|\lambda)\|^2 - \|(I - P_W)y\|^2.$$

Given a certain data set, we can use numerical methods to compute the supremum of these "norm" terms, as the supremum is well-defined. We shall denote that

$$K_u = \sup_{\lambda \in \mathbb{R}_+^{r+1}} \|E(u|\lambda)\|^2 \quad \text{and} \quad K_\theta = \sup_{\lambda \in \mathbb{R}_+^{r+1}} \|W(E(\theta|\lambda) - \hat{\theta})\|^2.$$

Now, we shall prove Proposition 6.

Proof of Proposition 6. Recall that $v(\theta) = \alpha\|W(\theta - \hat{\theta})\|^2 + \|u\|^2$. We shall prove that $v(\theta)$ satisfies drift condition. By law of iterated expectation, it is clear that

$$E[v(\theta)|\theta'] = E\big[E[v(\theta)|\lambda, \theta']|\,\theta'\big].$$

As before, we shall focus on the innermost expectation for now. Notice that the conditional density of $\theta|\lambda$ does not depend on θ'. By equations (3.1) and (3.2), we shall write

$$\begin{aligned}E[v(\theta)|\lambda,\, \theta'] &= E[v(\theta)|\lambda\,] \\ &= E[\alpha\|W(\theta - \hat{\theta})\|^2|\lambda] + E[\|u\|^2|\lambda] \\ &= \alpha\, \text{tr}(W\text{Var}(\theta|\lambda)W^T) + \alpha K_u + \text{tr}(Q_\lambda^{-1}) + K_\theta\end{aligned}$$

Now, we shall apply Lemma 3 and Proposition 7. Notice that rank$(Z) = q$, as Z has full column rank. Then, we have

$$E[v(\theta)|\lambda\,] < \alpha(p+q)\lambda_e^{-1} + \alpha K_u + \gamma_1\lambda_e^{-1} + \gamma_2 + K_\theta.$$

Recall that

$$E(\lambda_e^{-1}|\theta) = \frac{\|W(\theta - \hat{\theta})\|^2}{2a_e + N - 2} + \frac{\|(I - P_W)y\|^2 + 2b_e}{2a_e + N - 2}.$$

Now, we compute the outer expectation. We have

$$E[v(\theta)|\theta'] < \rho\alpha\|W(\theta' - \hat{\theta})\|^2 + L = \rho v(\theta') + L\ ,$$

where

$$\rho = \frac{(p+q) + \alpha^{-1}\gamma_1}{2a_e + N - 2} \quad \text{and} \quad L = (\alpha p + \alpha q + \gamma_1)\frac{\|(I - P_W)y\|^2 + 2b_e}{2a_e + N - 2} + \alpha K_u + \gamma_2 + K_\theta.$$

Since $a_e > \frac{1}{2}(2 + p + q - N)$, it is clear that $2a_e + N - 2 > 0$ and $\rho > 0$. To ensure that $\rho < 1$, we need

$$\alpha > \frac{\gamma_1}{2a_e + N - 2 - p - q}.$$

Since γ_1 is finite, we can always find some α such that $\rho \in [0, 1)$. This concludes our proof and we have established a drift condition with a "Type 2" bound. $\square$

3.3 Minorization Condition

Recall that $\lambda = (\lambda_e, \lambda_{u_1}, \dots, \lambda_{u_r})^T$ and $\theta = (\beta^T u^T)^T$. Let $k(\theta, \lambda|\theta'\lambda')$ be the Markov transition function for the chain of $\{(\theta^{(m)}, \lambda^{(m)})\}_{m=0}^{\infty}$. We know that probability transition kernel is

$$k(\theta, \lambda|\theta'\lambda') = f(\theta|\lambda) f(\lambda|\theta') \ .$$

Recall that our drift function is defined as

$$v(\theta) = \alpha\|W(\theta - \hat{\theta})\|^2 + \|u\|^2 \quad \text{for some constant } \alpha.$$

We define

$$S_\theta := \{\theta' : \alpha\|W(\theta' - \hat{\theta})\|^2 + \|\tilde{u}\|^2 \leqslant \delta\}.$$

Note that for any $\theta' \in S_\theta$, we have

$$f(\theta|\lambda) f(\lambda|\theta') \geqslant f(\theta|\lambda) \inf_{\theta' \in S_\theta} f(\lambda|\theta')$$

Recall that

$$\lambda_e|\theta, y \sim \text{Gamma}\left(a_e + \frac{N}{2}, b_e + \frac{\|(I - P_W)y\|^2 + \|W(\theta - \hat{\theta})\|^2}{2}\right),$$

and

$$\lambda_{u_i}|\theta, y \sim \text{Gamma}\left(a_i + \frac{q_i}{2}, b_i + \frac{\|u_i\|^2}{2}\right),$$

where $W = (X\ Z)$. As λ_e and all λ_{u_i} are independent, we have

$$\inf_{\theta' \in S_\theta} f(\lambda|\theta') = \inf_{\theta' \in S_\theta} \left[f(\lambda_e|\theta') \cdot \prod_{i=1}^{r} f(\lambda_{u_i}|\theta') \right].$$

Let

$$C_{\theta_e} := \{\theta' : \alpha\|W(\theta' - \hat{\theta})\|^2 \leqslant \delta\}, \text{ and } C_{\theta_i} := \{\theta' : \|\tilde{u}\|^2 \leqslant \delta\} \text{ for all } i \in \{1, 2, \dots, r\}.$$

Define $C_\theta = (\bigcap_{i=1}^{r} C_{\theta_i}) \cap C_{\theta_e}$. Then, it is clear that $C_\theta \supset S_\theta$. Therefore,

$$\begin{aligned}
\inf_{\theta' \in S_\theta} \left[f(\lambda_e|\theta') \cdot \prod_{i=1}^{r} f(\lambda_{u_i}|\theta') \right] &\geqslant \inf_{\theta' \in C_\theta} \left[f(\lambda_e|\theta') \cdot \prod_{i=1}^{r} f(\lambda_{u_i}|\theta') \right] \\
&\geqslant \inf_{\theta' \in C_\theta} f(\lambda_e|\theta') \cdot \prod_{i=1}^{r} \inf_{\theta' \in C_\theta} f(\lambda_{u_i}|\theta') \\
&\geqslant \inf_{\theta' \in C_{\theta_e}} f(\lambda_e|\theta') \cdot \prod_{i=1}^{r} \inf_{\theta' \in C_{\theta_i}} f(\lambda_{u_i}|\theta').
\end{aligned}$$

Hence, we have

$$f(\theta|\lambda) f(\lambda|\theta') \geqslant f(\theta|\lambda) \inf_{\theta' \in C_{\theta_e}} f(\lambda_e|\theta') \cdot \prod_{i=1}^{r} \inf_{\theta' \in C_{\theta_i}} f(\lambda_{u_i}|\theta') \ .$$

Let

$$g_e(\lambda_e) = \inf_{\theta' \in C_{\theta_e}} f(\lambda_e|\theta'),$$

and, for all $i \in \{1, 2, \dots, r\}$

$$g_i(\lambda_i) = \inf_{\theta' \in C_{\theta_i}} f(\lambda_{u_i}|\theta').$$

By Lemma 6 from Appendix A, it is clear that

$$g_e = \begin{cases} \text{Gamma}\left(a_e + \frac{N}{2}, b_e + \frac{\|(I - P_W)y\|^2}{2}\right) & \lambda_e \leqslant \lambda_e^* \\ \text{Gamma}\left(a_e + \frac{N}{2}, b_e + \frac{\|(I - P_W)y\|^2 + \alpha^{-1}\delta}{2}\right) & \lambda_e > \lambda_e^*, \end{cases} \tag{3.5}$$

and, for all $i \in \{1, 2, \ldots, r\}$

$$g_i = \begin{cases} \text{Gamma}\left(a_i + \frac{q_i}{2}, b_i\right) & \lambda_{u_i} \leqslant \lambda^*_{u_i} \\ \text{Gamma}\left(a_i + \frac{q_i}{2}, b_i + \frac{\delta}{2}\right) & \lambda_{u_i} > \lambda^*_{u_i}, \end{cases} \tag{3.6}$$

where

$$\lambda^*_e = \frac{(2a_e + N)}{\alpha^{-1}\delta} \log\left(1 + \frac{\alpha^{-1}\delta}{2b_e + \|(I_n - P_W)y\|^2}\right),$$

and, for all $i \in \{1, 2, \ldots, r\}$

$$\lambda^*_{u_i} = \frac{2a_i + q_i}{\delta} \log\left(1 + \frac{\delta}{2b_i}\right).$$

Put

$$\begin{aligned} \epsilon &= \int_{\mathbb{R}^{p+q}} \int_{\mathbb{R}^{r+1}_+} f(\theta|\lambda) \inf_{\theta' \in C_{\lambda_e}} f(\lambda_e|\theta') \cdot \prod_{i=1}^{r} \inf_{\theta' \in C_{\theta_i}} f(\lambda_{u_i}|\theta') d\lambda \, d\theta \\ &= \int_{\mathbb{R}_+} g_e(\lambda_e) d\lambda_e \cdot \prod_{i=1}^{r} \int_{\mathbb{R}_+} g_i(\lambda_i) d\lambda_i. \end{aligned}$$

The computation can be done through integrating $r + 1$ piecewise functions. Thus, the minorization condition is satisfied with the ϵ above, and it is easy to check that

$$q(\theta) = \epsilon^{-1} f(\theta|\lambda) \, g_e(\lambda_e) \prod_{i=1}^{r} g_i(\lambda_i) d\lambda$$

is a probability density function. We have established the minorization condition for drift function $v(\theta)$.

Remark 4. *Notice that this specific minorization condition is associated to our drift function $v(\theta)$. If we use another drift function, the minorization will change accordingly. Generally speaking, in order to have a good result for ϵ, we need to choose a drift function and bound it in the way that minimizes δ. We choose the drift function $v(\theta)$ over $v'(\theta)$ because we do not have the constant term $\|(I - P_W)\boldsymbol{y}\|^2$ as a part of δ.*

3.4 Example: Bounds on the Total Variation Distance

To help readers understand the performance of our bound, we shall provide a solid example and apply Theorem 1 to compute total variation distance. As in Chapter 2, we use the NBA 2015 data set and consider the logarithm of players' average points per game following a normal distribution. In addition to the linear regression coefficients β_1 and β_2, we also sort the players by their teams and consider the team effect u_i as the random effect for each team i. Particularly, we consider

$$\log(\text{PPG})_{ij} | \beta, u, \lambda \sim \text{N}\left(\beta_1 + \beta_2 \text{MIN}_{ij} + u_i, \ \lambda_e^{-1}\right),$$

where $\beta = (\beta_1, \beta_2)^T$ and $\lambda = (\lambda_e, \lambda_u)^T$. Notice that we only have one random effect in this model. Sometimes, people refer to this model as the random intercept model, as we can consider $\beta_1 + u_i$ as the random intercept of a certain player in team i.

In this particular example, we have sample size $N = 484$, and matrices X and Z have full column rank with $p = 2$ and $q = 30$. For prior distributions, we have

$$\beta \sim \text{N}(\mu_\beta, \Sigma_\beta) \perp \lambda_e \sim \text{Gamma}(13.17, \ 0.958),$$

where

$$\mu_\beta = \begin{pmatrix} 0.365 \\ 0.0733 \end{pmatrix} \quad \text{and} \quad \Sigma_\beta = \begin{bmatrix} 0.614 & -0.0216 \\ -0.0216 & 0.00835 \end{bmatrix}.$$

These two prior distributions are given based on some experts who study the data of NBA. Notice that we do not specify the prior for λ_u, because the prior for the team effect is not entirely clear. In this paper, we consider three different priors for λ_u to illustrate the effect of the random effect prior on the total variation distance.

We first consider $\lambda_u \sim \text{Gamma}(10, \ 20)$. Based on trials and errors, we pick $\iota = 3.44$ to obtain relatively small ρ and L, and we have $\gamma_1 = 7.524$ and $\gamma_2 = 2.187$. In the future, one may design an algorithm to optimize ι, but such algorithm must also take the optimization of α, κ and r into account.

In this example, the key to obtain decent total variation distance is to obtain the best ϵ from the minorization condition. From our experience, when we use a "Type 2" bound in the drift condition, the value of ϵ usually has significant influence on the total variation bound. With the values of γ_1 and γ_2, one can apply proposition 6 and compute ρ and L.

Then, we shall consider the value of ϵ as a function of α. To obtain the largest ϵ possible, we shall use the function `optimize` in `R` to select a value for α. In this example, we have $\alpha = 1.91$. Then, we may use the two-dimensional optimization function to obtain the best values for κ and r. Finally, one can calculate the smallest number of iteration such that the total variation distance is less than 0.01.

While keeping other priors the same, we shall also consider two other cases where $\lambda_u \sim \text{Gamma}(10,\ 50)$ or $\lambda_u \sim \text{Gamma}(5,\ 10)$. We shall follow the exact procedure described above to obtain the smallest number of iteration, m. We summarize these two cases in the following table:

Prior for λ_u	ι	α	ϵ	κ	r	m
$\text{Gamma}(10,\ 20)$	3.44	1.91	1.8×10^{-4}	3.10	0.021	1.22×10^{6}
$\text{Gamma}(10,\ 50)$	3.5	3.00	8.0×10^{-4}	2.33	0.020	2.55×10^{5}
$\text{Gamma}(5,\ 10)$	3.44	1.52	4.7×10^{-5}	2.33	0.020	5×10^{6}

From the table, we notice that ϵ is very small in all three cases, and such a small value imposes a strong restriction on the performance of total variation distance. If we continue to decrease the shape and the rate for random effect prior, ϵ will become too small for us to obtain any workable total variation distance.

The most helpful way to improve the total variation bound is to construct a "Type 3" bound by bounding the "norm" terms by a function of λ rather than bounding them by their supremum. Meanwhile, it is also valuable to study the optimization of ι and α that will yield a systematic approach to generate the best values for the given data set and priors.

Appendix A

Lemmas for 1-Sample Normal Model

A.1 Moments of a Gamma Distribution

Lemma 5 (Moments of a Gamma Distribution)**.** *Let X be a random variable of Gamma distribution with shape α and rate β. Then*

$$\mathbb{E}\left[X^t\right] = \beta^{-t}\frac{\Gamma(\alpha+t)}{\Gamma(\alpha)}$$

for all $t \in \mathbb{R}$ such that $\alpha + t > 0$. Particularly, if $a > 1$, then

$$\mathbb{E}\left[X^{-1}\right] = \frac{\beta}{\alpha-1}.$$

Proof. By the definition of moments, we have

$$\begin{aligned}\mathbb{E}\left[X^t\right] &= \int_{\mathbb{R}} x^t \frac{\beta^\alpha}{\Gamma(a)} x^{\alpha-1} e^{-\beta x}\, 1_{(0,\infty)} dx \\ &= \frac{\beta^\alpha}{\Gamma(\alpha)} \int_0^\infty x^{\alpha+t-1} e^{-\beta x} dx.\end{aligned}$$

If $\alpha + t > 0$, we may evaluate this integral by the properties of gamma density function. Hence, we conclude that

$$\mathbb{E}\left[X^t\right] = \frac{\beta^\alpha}{\Gamma(\alpha)}\frac{\Gamma(\alpha+t)}{\beta^{\alpha+t}} = \beta^{-t}\frac{\Gamma(\alpha+t)}{\Gamma(\alpha)}.$$

□

A.2 Infimum of Gamma Lemma

We note that this proof here is based off of the ideas of Jones and Hobert (2001).

Lemma 6. *Suppose that X is a random variable and*

$$X \sim \textit{Gamma}\left(\alpha, \beta+\gamma z\right) \quad \textit{for some constants } \alpha, \beta, \gamma > 0.$$

Define $C = \{z : c \leqslant z \leqslant d\}$. Let $f(x)$ denotes the probability density function of X. Then

$$\begin{aligned}\inf_{z\in C} f(x) &= \textit{min}\left(\textit{Gamma}(\alpha, \beta+\gamma c), \textit{Gamma}(\alpha, \beta+\gamma d)\right) \\ &= \begin{cases}\textit{Gamma}(\alpha, \beta+\gamma c) & x \leqslant x^* \\ \textit{Gamma}(\alpha, \beta+\gamma d) & x > x^*\end{cases}\end{aligned}$$

where

$$x^* = \frac{\alpha}{\gamma(d-c)} \log\left(\frac{\beta+\gamma d}{\beta+\gamma c}\right).$$

Proof. For fixed $x > 0$, consider $z \in [c, d]$ as a variable. Then, the density function of X becomes a function of z. We can write

$$h(z) = \frac{(\beta + \gamma z)^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-(\beta+\gamma z)x}.$$

To optimize function $h(z)$, we shall compute the first derivative of h. By the chain rule, we have

$$\begin{aligned}\frac{dh}{dz} &= \frac{x^{\alpha-1}}{\Gamma(\alpha)} e^{-(\beta+\gamma z)x} \left[\alpha\gamma(\beta + \gamma z)^{\alpha-1} - \gamma x(\beta + \gamma z)^\alpha\right] \\ &= \frac{x^{\alpha-1}}{\Gamma(\alpha)} e^{-(\beta+\gamma z)x} \gamma(\beta + \gamma z)^{\alpha-1} \left[\alpha - x(\beta + \gamma z)\right] \overset{\text{set}}{=} 0.\end{aligned}$$

As all terms outside the square bracket are positive, we can easily solve for z and conclude that

$$z^* = \frac{1}{\gamma}\left(\frac{\alpha}{x} - \beta\right).$$

This is the only critical point of the function $h(z)$. It is clear that

$$\frac{dh}{dz}\Big|_{z=z^*_-} > 0 \quad \text{and} \quad \frac{dh}{dz}\Big|_{z=z^*_+} < 0,$$

which implies that z^* is a local maximal point. Hence, for $z \in C$, the global minimal point of $h(z)$ is reached when $z = c$ or $z = d$. That's,

$$\inf_{z\in C} f(x) = \min\left(\text{Gamma}(\alpha, \beta + \gamma c), \text{Gamma}(\alpha, \beta + \gamma d)\right).$$

To write as a piecewise function, we need to calculate for what values of x such that $\text{Gamma}(\alpha, \beta + \gamma c) \leqslant \text{Gamma}(\alpha, \beta + \gamma d)$. We have

$$\begin{aligned} & \text{Gamma}(\alpha, \beta + \gamma c) \leqslant \text{Gamma}(\alpha, \beta + \gamma d) \\ \text{if and only if} \quad & \frac{(\beta + \gamma c)^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-(\beta+\gamma c)x} \leqslant \frac{(\beta + \gamma d)^\alpha}{\Gamma(a)} x^{\alpha-1} e^{-(\beta+\gamma d)x} \\ \text{if and only if} \quad & (\beta + \gamma c)^\alpha e^{-\gamma c x} \leqslant (\beta + \gamma d)^\alpha e^{-\gamma d x} \\ \text{if and only if} \quad & \alpha \log(\beta + \gamma c) - \gamma c x \leqslant \alpha \log(\beta + \gamma d) - \gamma d x \\ \text{if and only if} \quad & x \leqslant \frac{\alpha}{\gamma(d - c)} \log\left(\frac{\beta + \gamma d}{\beta + \gamma c}\right). \end{aligned}$$

We define $x^* = \frac{\alpha}{\gamma(d-c)} \log\left(\frac{\beta+\gamma d}{\beta+\gamma c}\right)$. Hence, we conclude that

$$\inf_{z\in C} f(x) = \begin{cases} \text{Gamma}(\alpha, \beta + \gamma c) & x \leqslant x^* \\ \text{Gamma}(\alpha, \beta + \gamma d) & x > x^* \end{cases}.$$

□

Appendix B

Lemmas for the Linear Regression Model

B.1 Reparameterization Lemma

Lemma 7. *If we denote the Gibbs sampler for the reparameterized linear regression model $B^{(m)}$ and the Gibbs sampler for the original model $A^{(m)}$ then the following equality holds*

$$\|P_A^{(m)}(x,\cdot) - \Pi(\cdot)\| = \|P_B^{(m)}(x,\cdot) - (f_*\Pi)(\cdot)\|.$$

where $\Pi(\cdot)$ is the invariant measure of the chain $A^{(m)}$ and $(f_*\Pi)(\cdot)$ is the invariant measure of the chain $B^{(m)}$. This essentially means that the rate at which the two chains converge to their stationary distributions is the same.

Proof of Lemma 7. Here we shall need to use several results from Roberts and Rosenthal (2001). What we want to use is their Corollary 2 which states:

Corollary 1. *Let $\{X^{(m)}\}$ be a Markov chain with stationary distribution $\Pi(\cdot)$. Let $Y^{(m)} = f(X^{(m)})$ for some measurable function f, and suppose that $Y^{(m)}$ is Markovian and de-initializing for $X^{(m)}$. Then*

$$\|P_X^{(m)}(x|\cdot) - \Pi(\cdot)\|_{TV} = \|P_Y^{(m)}(x,\cdot) - (f * \Pi)(\cdot)\|_{TV}$$

By Lemma 1 from their paper if we can show that our transformation is a deterministic measurable function f such that $f(B^{(m)}) = A^{(m)}$ then we know that $\{B^{(m)}\}$ is de-initializing for$\{A^{(m)}\}$. If f is one-to-one then we also know that $B^{(m)} = f^{-1}(A^{(m)})$ which allows us to say that $B^{(m)} = f^{-1}(A^{(m)})$ where f^{-1} is a measurable function. We also know that $\{B^{(m)}\}$ is Markovian as it is also a Gibbs sampler. This would allow us to apply Corollary 2 which gives us exactly the equality from Lemma 7.

Let us consider the Gibbs sampler for the original model as having draws $(\beta^{\tilde{(m)}}, \tau^{\tilde{(m)}})$. We shall consider the Gibbs sampler for the reparameterized model as having draws $(\beta^{(m)}, \tau^{(m)})$. We recall that the new model may be written as

$$Y|\beta, \sigma \sim N_n(X\beta, I_n\sigma^2),$$
$$\beta \sim N_p(\vec{0}, \Sigma_\beta), \quad \perp \quad \tau \sim \text{Gamma}(a, b)\ .$$

It is clear that the Gibbs sampler for this new model has $\beta^{(m)} = \tilde{\beta}^{(m)} - \mu_\beta$ and $\tau^{(m)} = \tilde{\tau}^{(m)}$ for all m. We see that the transformation from the original chain to the other is just $f(\tilde{\beta}^{(m)}, \tilde{\tau}^{(m)}) = (\tilde{\beta}^{(m)} - \mu_\beta, \tilde{\tau}^{(m)})$. This transformation is clearly deterministic, measurable, and one-to-one thus we satisfy the properties necessary to apply corollary 2, completing our proof.

□

B.2 Monotone Nonincreasing Matrix Functions

To prove a function g is non-increasing, we shall show that its first derivative is negative wherever the function is defined. In our derivation, we shall some use techniques about differentiation of matrices of functions. Let's first recall some important results in matrix derivatives.

B.2.1 Review of Matrix Derivatives

Lemma 8 (Lemma 15.4.2 by Harville (1997))**.** *Let $\boldsymbol{F} = \{f_{is}\}$ and $\boldsymbol{G} = \{g_{is}\}$ represent $p \times q$ matrices of functions, defined on a set S, of a vector $\boldsymbol{x} = \left(x_1, x_2, \ldots, x^{(m)}\right)^T$ of m variables. And, let a and b represent constants or (more generally) functions (defined on S) that are continuous at every interior point of S and are such that $a(\boldsymbol{x})$ and $b(\boldsymbol{x})$ do not vary with x_j. Then, at any interior point $\mathbf{c}$ (of S) at which $\boldsymbol{F}$ and $\boldsymbol{G}$ are continuously differentiable, $a\boldsymbol{F} + b\boldsymbol{G}$ is continuously differentiable and*

$$\frac{\partial(a\boldsymbol{F} + b\boldsymbol{G})}{\partial x_j} = a\frac{\partial \boldsymbol{F}}{\partial x_j} + b\frac{\partial \boldsymbol{G}}{\partial x_j}. \tag{B.1}$$

Lemma 9 (Lemma 15.4.3 by Harville (1997))**.** *Let $\boldsymbol{F} = \{f_{is}\}$ and $\boldsymbol{G} = \{g_{is}\}$ represent $p \times q$ and $q \times r$ matrices of functions, defined on a set S, of a vector $\boldsymbol{x} = \left(x_1, x_2, \ldots, x^{(m)}\right)^T$ of m variables. Then, at any interior point $\mathbf{c}$ (of S) at which $\boldsymbol{F}$ and $\boldsymbol{G}$ are continuously differentiable, $\boldsymbol{FG}$ is continuously differentiable and*

$$\frac{\partial \boldsymbol{FG}}{\partial x_j} = \boldsymbol{F}\frac{\partial \boldsymbol{G}}{\partial x_j} + \frac{\partial \boldsymbol{F}}{\partial x_j}\boldsymbol{G}. \tag{B.2}$$

Remark 5. *In the special case where (for $\boldsymbol{x} \in S$) $\boldsymbol{F}(\boldsymbol{x})$ is constant or (more generally) does not vary with x_j, formula $(B.2)$ simplifies to*

$$\frac{\partial \boldsymbol{FG}}{\partial x_j} = \boldsymbol{F}\frac{\partial \boldsymbol{G}}{\partial x_j}. \tag{B.3}$$

And, in the special case where (for $\boldsymbol{x} \in S$) $\boldsymbol{G}(\boldsymbol{x})$ is constant or (more generally) does not vary with x_j, formula $(B.2)$ simplifies to

$$\frac{\partial \boldsymbol{FG}}{\partial x_j} = \frac{\partial \boldsymbol{F}}{\partial x_j}\boldsymbol{G}. \tag{B.4}$$

The results of Lemma 9 can be extended (by repeated application) to the product of three or more matrices.

Lemma 10. *Let $\boldsymbol{F}, \boldsymbol{G}$, and $\boldsymbol{H}$ represent $p \times q$, $q \times r$, and $r \times v$ matrices of functions, defined on a set S, of a vector $\boldsymbol{x} = \left(x_1, x_2, \ldots, x^{(m)}\right)^T$ of m variables. Then, at any interior point (of S) at which $\boldsymbol{F}$, $\boldsymbol{G}$, and $\boldsymbol{H}$ are continuously differentiable, $\boldsymbol{FGH}$ is continuously differentiable and*

$$\frac{\partial \boldsymbol{FGH}}{\partial x_j} = \boldsymbol{FG}\frac{\partial \boldsymbol{H}}{\partial x_j} + \boldsymbol{F}\frac{\partial \boldsymbol{G}}{\partial x_j}\boldsymbol{H} + \frac{\partial \boldsymbol{F}}{\partial x_j}\boldsymbol{GH}. \tag{B.5}$$

Remark 6. *In the special case where (for $\boldsymbol{x} \in S$) $\boldsymbol{F}(\boldsymbol{x})$ and $\boldsymbol{H}(\boldsymbol{x})$ are constant or (more generally) do not vary with x_j, formula $(B.5)$ simplifies to*

$$\frac{\partial \boldsymbol{FGH}}{\partial x_j} = \boldsymbol{F}\frac{\partial \boldsymbol{G}}{\partial x_j}\boldsymbol{H}. \tag{B.6}$$

We also want to include one helpful result about differentiation of a trace of a matrix.

Lemma 11. *Let $\boldsymbol{F} = \{f_{is}\}$ represent a $p \times p$ matrix of functions, defined on a set S, of a vector $\boldsymbol{x} = \left(x_1, x_2, \ldots, x^{(m)}\right)^T$ of m variables. Then, at any interior point $\mathbf{c}$ (of S) at which $\boldsymbol{F}$ is continuously differentiable, tr$(\boldsymbol{F})$ is continuously differentiable and*

$$\frac{\partial tr(\boldsymbol{F})}{\partial x_j} = tr\left(\frac{\partial \boldsymbol{F}}{\partial x_j}\right) \tag{B.7}$$

Finally, we include one property of derivatives of inverse matrices.

Lemma 12. *Let $\boldsymbol{F} = \{f_{is}\}$ represent a $p \times p$ matrix of functions of a vector $\boldsymbol{x} = \left(x_1, x_2, \ldots, x^{(m)}\right)^T$ of m variables. Suppose that S is the set of all $\boldsymbol{x}$-values for which $\boldsymbol{F}(\boldsymbol{x})$ is nonsingular or is a subset of that set. Denote by $\mathbf{c}$ any interior point (of S) at which $\boldsymbol{F}$ is continuously differentiable. Then, $\boldsymbol{F}^{-1}$ is continuously differentiable at $\mathbf{c}$. And,*

$$\frac{\partial \boldsymbol{F}^{-1}}{\partial x_j} = -\boldsymbol{F}^{-1}\frac{\partial \boldsymbol{F}}{\partial x_j}\boldsymbol{F}^{-1}. \tag{B.8}$$

Now, with these tools, we believe readers are ready to tackle the problems in the next section.

B.2.2 Results using Matrix Calculus

Recall that

$$\Psi_\tau = \left(\tau X^T X + \Sigma_\beta^{-1}\right)^{-1} \text{ is a positive-definite matrix.}$$

To calculate the first derivative of Ψ_τ, we denote

$$Q_\tau = \Psi_\tau^{-1} = (\tau X^T X + \Sigma_\beta^{-1}).$$

It follows from the definition of derivative that Q_τ is continuously differentiable with respect to τ, as $X^T X$ and Σ_β^{-1} are constant. Then, it follows from Lemma 12 that Ψ_τ is continuously differentiable. By Lemmas 8 and 12, it is easy to show that

$$\frac{\partial \Psi_\tau}{\partial \tau} = \frac{\partial Q_\tau^{-1}}{\partial \tau} = -Q_\tau^{-1}\frac{\partial Q_\tau}{\partial \tau}Q_\tau^{-1} = -\Psi_\tau \frac{\partial\left(\tau X^T X + \Sigma_\beta^{-1}\right)}{\partial \tau}\Psi_\tau = -\Psi_\tau X^T X \Psi_\tau. \tag{B.9}$$

We define

$$A_\tau = X\Psi_\tau X^T \tag{B.10}$$

Notice that A_τ is non-negative define (i.e. positive semi-definite). By Lemma 9 and Remark 5, it is clear that A_τ is continuously differentiable. Then, we have

$$\frac{\partial A_\tau}{\partial \tau} = X\frac{\partial \Psi_\tau}{\partial \tau}X^T = X\left(-\Psi_\tau X^T X \Psi_\tau\right)X^T = -A_\tau^2, \tag{B.11}$$

By Lemma 9, we can also compute that

$$\frac{\partial(\tau A_\tau)}{\partial \tau} = -\tau A_\tau^2 + A_\tau. \tag{B.12}$$

Before we start our main problem, we shall state some important properties about generalized matrix inverse and projection matrices, as we do not assume that X has full rank in our theorem.

B.2.3 Generalized Inverse and Projection Matrices

For any matrix $\boldsymbol{X}$, we define

$$\boldsymbol{P}_X = X\left(X^T X\right)^- X^T. \tag{B.13}$$

Note that one can show that $\boldsymbol{P}_X$ is invariant to the choice of the generalized inverse $(X^T X)^-$. For any matrix $\boldsymbol{X}$, it is clear that $\boldsymbol{P}_X$ is the projection matrix for $\mathcal{C}(\boldsymbol{X})$. Then, we state the following properties for projection matrices.

Lemma 13 (Lemma 12.3.4 by Harville (1997)). *Let $\boldsymbol{X}$ represent any $n \times p$ matrix. Then, $\boldsymbol{P}_X\boldsymbol{X} = \boldsymbol{X}$; that is, $X(X^T X)^- X^T X = X$; that is, $(X^T X)^- X^T$ is a generalized inverse of $\boldsymbol{X}$.*

Remark 7. *For Moore-Penrose pseudoinverse, it follows from Lemma 13 that*

$$(X^T X)^+ X^T = X^+. \tag{B.14}$$

Lemma 14 (Theorem 12.3.5 by Harville (1997)). *Let $\boldsymbol{X}$ represent an $n \times p$ matrix, and let $\boldsymbol{W}$ represent any $n \times q$ matrix such that $\mathcal{C}(\boldsymbol{W}) \subset \mathcal{C}(\boldsymbol{X})$. That is, there exists a matrix $\boldsymbol{F}$ such that $\boldsymbol{W} = \boldsymbol{X}\boldsymbol{F}$. Then,*

$$\boldsymbol{P}_X\boldsymbol{W} = \boldsymbol{W}, \text{ and } \boldsymbol{W}^T\boldsymbol{P}_X = \boldsymbol{W}^T. \tag{B.15}$$

Now, equipped with all the powerful results stated above, we are ready to move to the main lemma.

B.2.4 Norm Lemma

Recall that, for all $\tau \geqslant 0$,

$$g(\tau) := \left\|\tau A_\tau y - X\hat{\beta}\right\|^2, \tag{B.16}$$

We shall prove that $g(\tau)$ is monotone nonincreasing and convex for all τ.

Proof of Lemma 1. It follows from the Section $B.2.2$ that $g(\tau)$ is continuously differentiable with respect to τ. We shall prove that $g(\tau)$ is non-increasing by showing that its first derivative is non-positive for all $\tau \geqslant 0$. By the definition of Frobenius Norm, we have

$$g(\tau) = \text{tr}\left[\left(\tau A_\tau y - X\hat{\beta}\right)^T \left(\tau A_\tau y - X\hat{\beta}\right)\right]. \tag{B.17}$$

We can expand the terms inside the square bracket and apply the properties of the trace. Then, we have

$$g(\tau) = \tau^2 \cdot \text{tr}\left[(A_\tau y)^T (A_\tau y)\right] + \text{tr}\left[(X\hat{\beta})^T (X\hat{\beta})\right] - 2\tau \cdot \text{tr}\left[(A_\tau y)^T (X\hat{\beta})\right]$$

We note that each term inside the square brackets is a scalar. Thus, we can drop the trace functions and work directly with the matrix products. Our next step is to differentiate each of these terms with respect to τ. It is clear that $(X\hat{\beta})^T(X\hat{\beta})$ is a constant, so its derivative with respect to τ is 0. Now, for the other terms, we shall compute the first derivative term by term. We shall call $g_1(\tau) := \tau^2 (A_\tau y)^T (A_\tau y) = \tau^2 (y^T A_\tau)(A_\tau y)$. Then, by Lemma 10 and formula $(B.11)$, we have

$$\begin{aligned} g_1'(\tau) &= \tau^2 (y^T A_\tau)\frac{\partial (A_\tau y)}{\partial \tau} + \tau^2 \frac{\partial (y^T A_\tau)}{\partial \tau}(A_\tau y) + \frac{\partial \tau^2}{\partial \tau}(y^T A_\tau)(A_\tau y) \\ &= \tau^2 (y^T A_\tau)(-A_\tau^2 y) + \tau^2(-y^T A_\tau^2)(A_\tau y) + 2\tau (y^T A_\tau)(A_\tau y) \\ &= -2\tau^2 y^T A_\tau^3 y + 2\tau y^T A_\tau^2 y \\ &= 2\tau y^T A_\tau^2 (-\tau A_\tau + I_n) y \\ &= 2\tau y^T A_\tau^2 (I_n - \tau A_\tau) y. \end{aligned} \tag{B.18}$$

We shall call $g_2(\tau) := -2\tau (A_\tau y)^T (X\hat{\beta}) = -2\tau (y^T A_\tau)(X\hat{\beta})$. Then, by formula $(B.12)$, we have

$$\begin{aligned} g_2'(\tau) &= -2y^T \frac{\partial (\tau A_\tau)}{\partial \tau} X\hat{\beta} \\ &= -2y^T(-\tau A_\tau^2 + A_\tau)X\hat{\beta} \\ &= -2y^T A_\tau (I_n - \tau A_\tau) X\hat{\beta}. \end{aligned} \tag{B.19}$$

Now, from equations $(B.18)$ to $(B.19)$, it is very clear that

$$\begin{aligned} g'(\tau) &= g_1'(\tau) + g_2'(\tau) \\ &= 2\tau y^T A_\tau^2 (I_n - \tau A_\tau) y - 2y^T A_\tau (I_n - \tau A_\tau) X\hat{\beta} \end{aligned} \tag{B.20}$$

We can rewrite $g_2'(\tau)$ by using

$$\hat{\beta} = (X^T X)^+ X^T y. \tag{B.21}$$

It is clear that A_τ commutes with $(I_n - \tau A_\tau)$. And, $\mathcal{C}(A_\tau) \subset \mathcal{C}(X)$, as $A_\tau = X(VX^T)$. For equation $(B.19)$, we shall use the formula in $(B.21)$ and apply properties of the projection matrix in $(B.13)$ and $(B.15)$. Then, we have

$$\begin{aligned} g_2'(\tau) &= -2y^T A_\tau (I_n - \tau A_\tau) X\hat{\beta} \\ &= -2y^T A_\tau (I_n - \tau A_\tau) X (X^T X)^+ X^T y \\ &= -2y^T A_\tau (I_n - \tau A_\tau) \boldsymbol{P}_X y \\ &= -2y^T (I_n - \tau A_\tau) A_\tau \boldsymbol{P}_X y \\ &= -2y^T (I_n - \tau A_\tau) A_\tau y \\ &= -2y^T A_\tau (I_n - \tau A_\tau) y. \end{aligned} \tag{B.22}$$

Now we can combine $g_1'(\tau)$ in (B.18) and $g_2'(\tau)$ in (B.22) by factoring out the common term $2y^T A_\tau (I_n - A_\tau)$ from the front and y from the back

$$\begin{aligned} g_1'(\tau) + g_2'(\tau) &= 2[y^T A_\tau (I_n - \tau A_\tau)(\tau A_\tau - I_n) y] \\ &= -2y^T A_\tau (I_n - \tau A_\tau)(I_n - \tau A_\tau) y \\ &= -2y^T (I_n - \tau A_\tau) A_\tau (I_n - \tau A_\tau) y \end{aligned} \tag{B.23}$$

Let $S_\tau := (I_n - \tau A_\tau)y$. We see that S_τ is a vector and (B.23) is of the form

$$-2S_\tau^T A_\tau S_\tau \ . \tag{B.24}$$

We recall though that A_τ is a nonnegative definite matrix so this term must be nonnegative. Thus:

$$g'(\tau) \leq 0 \text{ for all } \tau \in \mathbb{R}_+$$

which means that $g(\tau)$ is monotone nonincreasing for all values of τ.
To prove that $g(\tau)$ is convex we shall first calculate the derivative of S_τ.

$$\begin{aligned}\frac{\partial S_\tau}{\partial \tau} &= \left(I_n - \frac{\partial \tau A_\tau}{\partial \tau}\right) y \\ &= (\tau A_\tau^2 - A_\tau)y \\ &= A_\tau(\tau A_\tau - I_n)y \\ &= -A_\tau S_\tau.\end{aligned} \tag{B.25}$$

Applying (B.25) we may easily calculate $g''(\tau)$:

$$\begin{aligned}g''(\tau) &= -2\frac{\partial S_\tau^T}{\partial \tau} A_\tau S_\tau - 2S_\tau^T \frac{\partial A_\tau}{\partial \tau} S_\tau - 2S_\tau A_\tau \frac{\partial S_\tau}{\partial \tau} \\ &= 2S_\tau^T A_\tau A_\tau S_\tau + 2S_\tau^T A_\tau^2 S_\tau + 2S_\tau^T A_\tau A_\tau S_\tau \\ &= 6S_\tau^T A_\tau^2 S_\tau \ .\end{aligned} \tag{B.26}$$

Because A_τ is nonnegative definite it follows that A_τ^2 is too thus (B.26) must be nonnegative. This implies that $g(\tau)$ is convex for all values of $\tau \in \mathbb{R}_+$.

□

B.2.5 Trace Lemma

Here we shall also prove that the trace term that arises in the drift condition is nonincreasing and convex. We shall denote $h(\tau) := \text{tr}(X^T X \Psi_\tau)$ for ease of reading in this proof.

Proof of Lemma 2. We recall that the trace of a product is unchanged under cyclic permutations thus $\text{tr}(X^T XV) = \text{tr}(XVX^T)$. As before we will denote XVX^T as A_τ. We see that by equation B.11 and Lemma 11

$$h'(\tau) = -\text{tr}(A_\tau^2).$$

Note that A_τ is symmetric thus we may say that $-\text{tr}(A_\tau^2) = -\text{tr}(A_\tau A_\tau^T)$. In this form it is clear to see that each term in the trace is the standard inner product of a vector with itself and is thus nonnegative. This implies that $\text{tr}(A_\tau^2) \geq 0$ and $-\text{tr}(A_\tau^2) \leq 0$. We see then that $h(\tau)$ is monotone nonincreasing as its derivative is never positive. Next we shall take the second derivative of $h(\tau)$ to prove that it is convex. We see that by Lemma 9 we get

$$h''(\tau) = 2\text{tr}(A_\tau^3).$$

Recall that A_τ is symmetric thus by the spectral theorem we may write it as $A_\tau = QDQ^T$ where Q is an orthogonal matrix. Because A_τ is nonnegative definite we are also able to write $A_\tau^{3/2} = QD^{3/2}Q^T$ where $D^{3/2}$ is simply D with each diagonal entry raised to the 3/2 power. We see that $A_\tau^{3/2}A_\tau^{3/2} = A_\tau^3$ and $A_\tau^{3/2}$ is symmetric. We may then state

$$h''(\tau) = 2\text{tr}(A_\tau^{3/2}A_\tau^{3/2}) = 2\text{tr}(A_\tau^{3/2}{A_\tau^{3/2}}^T).$$

It is once again clear that each term of the trace is the standard inner product of a vector with itself making it nonnegative. This implies that the function $h''(\tau) \geq 0$ and thus $h(\tau)$ is convex. □

Appendix C

Lemmas for Linear Mixed Model

This first Lemma is borrowed from Román and Hobert (2015)

Lemma 15. *Suppose Ω is an $n \times n$ matrix of the form*

$$\Omega = A^T A v + \Upsilon,$$

where v is a positive constant, A is a non-null $m \times n$ matrix and Υ is an $n \times n$ diagonal matrix with positive diagonal elements, $\{v_i\}_{i=1}^n$. Let $O^T D O$ be the spectral decomposition of $A^T A$, so O is an n-dimensional orthogonal matrix, and D is a diagonal matrix whose diagonal elements, $\{d_i\}_{i=1}^n$, are eigenvalues of $A^T A$. Also, let $D^\perp$ denote the n-dimensional diagonal matrix whose diagonal elements, $\{d_i\}_{i=1}^n$, are given by

$$d_i^\perp = \begin{cases} 1, & d_i = 0, \\ 0, & d_i \neq 0. \end{cases}$$

Then

1. $\Omega^{-1} \preccurlyeq (A^T A)^+ v^{-1} + O^T D^\perp O v_{min}^{-1}$,
2. $tr(\Omega^{-1}) \leqslant tr((A^T A)^+ v^{-1} + (n - rank(A)) v_{min}^{-1}$,
3. $tr(A \Omega^{-1} A^T) \leqslant rank(A) v^{-1}$,

where $(A^T A)^+$ denotes the Moore-Penrose inverse of $A^T A$ and $v_{min} = min_{1 \leqslant i \leqslant n}\{v_i\}$.

Here we present a proof of Lemma 3 which uses the results of Lemma 15.

Proof of Lemma 3. By the properties of the trace, we can simplify the third term in (3.4) by

$$\begin{aligned}
&\text{tr}((I - M_\lambda) Z Q_\lambda^{-1} Z^T (I + M_\lambda)) \\
&= \text{tr}(Z Q_\lambda^{-1} Z^T - M_\lambda Z Q_\lambda^{-1} Z^T + Z Q_\lambda^{-1} Z^T M_\lambda - M_\lambda Z Q_\lambda^{-1} Z^T M_\lambda) \\
&= \text{tr}(Z Q_\lambda^{-1} Z^T) - \text{tr}(M_\lambda Z Q_\lambda^{-1} Z^T) + \text{tr}(Z Q_\lambda^{-1} Z^T M_\lambda) - \text{tr}(M_\lambda Z Q_\lambda^{-1} Z^T M_\lambda) \\
&= \text{tr}(Z Q_\lambda^{-1} Z^T) - \text{tr}(M_\lambda Z Q_\lambda^{-1} Z^T M_\lambda),
\end{aligned}$$

as $\text{tr}(M_\lambda Z Q_\lambda^{-1} Z^T) = \text{tr}(Z Q_\lambda^{-1} Z^T M_\lambda)$. Then, it is clear write

$$\begin{aligned}
\text{tr}(W \text{Var}(\theta|\lambda) W^T) &= \text{tr}(Z Q_\lambda^{-1} Z^T) + \text{tr}(X T_\lambda^{-1} X^T) - \text{tr}((I - M_\lambda) Z Q_\lambda^{-1} Z^T (I + M_\lambda)) \\
&= \text{tr}(Z Q_\lambda^{-1} Z^T) + \text{tr}(X T_\lambda^{-1} X^T) - [\, \text{tr}(Z Q_\lambda^{-1} Z^T) - \text{tr}(M_\lambda Z Q_\lambda^{-1} Z^T M_\lambda) \,] \\
&= \text{tr}(X T_\lambda^{-1} X^T) + \text{tr}(M_\lambda Z Q_\lambda^{-1} Z^T M_\lambda).
\end{aligned}$$

This is the first statement in Lemma 3.

Recall that $T_\lambda = \lambda_e X^T X + \Sigma_\beta^{-1} \succcurlyeq \lambda_e X^T X$. Then, if X has full column rank, it is clear that

$$T_\lambda^{-1} \preccurlyeq (\lambda_e X^T X)^{-1}.$$

Thus, we have

$$\text{tr}(XT_\lambda^{-1}X^T) \leqslant \text{tr}[\, X(\lambda_e X^T X)^{-1} X^T \,] = p\lambda_e^{-1},$$

where p is the column rank of matrix X. One can show that $M_\lambda = I - \lambda_e X T_\lambda^{-1} X^T$ is a non-negative definite matrix and we shall use Spectral decomposition to write $M_\lambda = \Gamma^T D_M \Gamma$ for some orthogonal matrix Γ and diagonal matrix D_M with positive elements. Define

$$M^{1/2} = \Gamma^T D_M^{1/2} \Gamma,$$

where $D_M^{1/2}$ is the diagonal matrix with each element equal to the square root of the each corresponding element in D_M. Then, we have

$$\text{tr}(M_\lambda Z Q_\lambda^{-1} Z^T M_\lambda) = \text{tr}(M_\lambda^{1/2} M_\lambda^{1/2} Z Q_\lambda^{-1} Z^T M_\lambda^{1/2} M_\lambda^{1/2}) = \text{tr}(M_\lambda^{1/2} H_\lambda M_\lambda^{1/2}),$$

where $H_\lambda = M_\lambda^{1/2} Z Q_\lambda^{-1} Z^T M_\lambda^{1/2}$. Notice that H_λ is non-negative definite and we can define $H_\lambda^{1/2}$ by the procedure above. Then, we have

$$\begin{aligned}\text{tr}(M_\lambda Z Q_\lambda^{-1} Z^T M_\lambda) &= \text{tr}(M_\lambda^{1/2} H_\lambda M_\lambda^{1/2}) \\ &= \text{tr}(M_\lambda^{1/2} H_\lambda^{1/2} H_\lambda^{1/2} M_\lambda^{1/2}) \\ &= \text{tr}(H_\lambda^{1/2} M_\lambda H_\lambda^{1/2}).\end{aligned}$$

Notice that in the last step, we use the cyclic property of the trace. Since $M_\lambda \preccurlyeq I$, we can write

$$\text{tr}(M_\lambda Z Q_\lambda^{-1} Z^T M_\lambda) = \text{tr}(H_\lambda^{1/2} M_\lambda H_\lambda^{1/2}) \leqslant \text{tr}(H_\lambda)$$

By the third statement of Lemma 15, we have

$$\text{tr}(H_\lambda) = \text{tr}(M_\lambda^{1/2} Z Q_\lambda^{-1} Z^T M_\lambda^{1/2}) \leqslant \text{rank}(M_\lambda^{1/2} Z)\lambda_e^{-1} = \text{rank}(Z)\lambda_e^{-1},$$

as $M_\lambda^{1/2}$ is an invertible matrix. Hence, we can conclude that

$$\text{tr}(W \text{Var}(\theta|\lambda) W^T) = \text{tr}(X T_\lambda^{-1} X^T) + \text{tr}(M_\lambda Z Q_\lambda^{-1} Z^T M_\lambda) \leqslant (p + \text{rank}(Z))\lambda_e^{-1}.$$

This completes the proof of the second statement in Lemma 3. □

Appendix D

Matrix Calculus Results for the Linear Mixed Model

Before we begin we shall first restate several definitions we presented earlier for ease of reading. Let $\lambda := (\lambda_e, \lambda_{u_1}, \ldots, \lambda_{u_r})^T$. Define $\Lambda_u = \oplus_{i=1}^r \lambda_{u_i} I_{q_i}$ and $q = q_1 + \cdots + q_r$. In order to simplify calculations, we define $T_\lambda = \lambda_e X^T X + \Sigma_\beta^{-1}$, $M_\lambda = I - \lambda_e X T_\lambda^{-1} X^T$, and $Q_\lambda = \lambda_e Z^T M_\lambda Z + \Lambda_u$. Therefore, it is clear that T_λ, M_λ, and Q_λ are differentiable with respect to λ_e and λ_{u_i}. We have

$$\frac{\partial T_\lambda}{\partial \lambda_e} = X^T X,$$

and

$$\frac{\partial T_\lambda^{-1}}{\partial \lambda_e} = -T_\lambda^{-1} X^T X T_\lambda^{-1}.$$

Then, we shall have

$$\frac{\partial M_\lambda}{\partial \lambda_e} = -X T_\lambda^{-1} X^T + \lambda_e X T_\lambda^{-1} X^T X T_\lambda^{-1} X^T. \tag{D.1}$$

From now, we denote

$$A_\lambda = X T_\lambda^{-1} X^T,$$

and notice that A_λ and M_λ commute. Then,

$$\frac{\partial M_\lambda}{\partial \lambda_e} = -A_\lambda + \lambda_e A_\lambda^2 = -A_\lambda M_\lambda,$$

and

$$\begin{aligned}\frac{\partial Q_\lambda}{\partial \lambda_e} &= Z^T M_\lambda Z - \lambda_e Z^T A_\lambda M_\lambda Z \\ &= Z^T M_\lambda (I_n - \lambda_e A_\lambda) Z \\ &= Z^T M_\lambda^2 Z, \end{aligned} \tag{D.2}$$

as A_λ commutes with M_λ. Thus, we have

$$\frac{\partial Q_\lambda^{-1}}{\partial \lambda_e} = -Q_\lambda^{-1} Z^T M_\lambda^2 Z Q_\lambda^{-1}. \tag{D.3}$$

Now, we are ready to prove Lemma 4.

Proof of Lemma 4.

$$\frac{\partial \, \text{tr}(Q_\lambda^{-1})}{\partial \lambda_e} = -\text{tr}(Q_\lambda^{-1} Z^T M_\lambda^2 Z Q_\lambda^{-1}) \, .$$

We recall that M_λ is a positive definite matrix, and we can write $M_\lambda = M_\lambda^{1/2} M_\lambda^{1/2}$. Then, we have

$$\frac{\partial \, \text{tr}(Q_\lambda^{-1})}{\partial \lambda_e} = -\text{tr}(Q_\lambda^{-1} Z^T M_\lambda^{1/2} M_\lambda M_\lambda^{1/2} Z Q_\lambda^{-1}) \, .$$

Notice that Q_λ, Q_λ^{-1}, and $M_\lambda^{1/2}$ are symmetric, so we have $Q_\lambda^{-1} Z^T M_\lambda^{1/2} = (M_\lambda^{1/2} Z Q_\lambda^{-1})^T$. It becomes clear that

$$\frac{\partial \, \text{tr}(Q_\lambda^{-1})}{\partial \lambda_e} = -\text{tr}[\, (M_\lambda^{1/2} Z Q_\lambda^{-1})^T M_\lambda (M_\lambda^{1/2} Z Q_\lambda^{-1}) \,].$$

Let $\{\phi_i\}_{i=1}^q$ denote the column vectors of $M_\lambda^{1/2} Z Q_\lambda^{-1}$. Now, we have

$$\frac{\partial \, \text{tr}(Q_\lambda^{-1})}{\partial \lambda_e} = -\text{tr}[\, (M_\lambda^{1/2} Z Q_\lambda^{-1})^T M_\lambda (M_\lambda^{1/2} Z Q_\lambda^{-1}) \,] = -\sum_{i=1}^{q} \phi_i^T M_\lambda \phi_i.$$

As M_λ is positive-definite, it is clear that $\phi_i^T M_\lambda \phi_i > 0$ for all $i \in \{1, 2, \ldots, q\}$. Therefore, we can conclude that

$$\frac{\partial \text{tr}(Q_\lambda^{-1})}{\partial \lambda_e} < 0 \,. \tag{D.4}$$

That is, tr(Q_λ^{-1}) is monotone decreasing with respect to λ_e. We shall now consider differentiating with respect to λ_{u_i}. We see that

$$\frac{\partial \, \text{tr}(Q_\lambda^{-1})}{\partial \lambda_{u_i}} = -\text{tr}(Q_\lambda^{-1} \Lambda_{q_i} Q_\lambda^{-1}) \,,$$

where Λ_{q_i} denotes the partial derivative of Λ_u with respect to λ_{u_i}. Let $\{\phi'_j\}_{j=1}^q$ denote the column vectors of Q_λ^{-1}. We may then state:

$$\frac{\partial \, \text{tr}(Q_\lambda^{-1})}{\partial \lambda_{u_i}} = -\text{tr}(Q_\lambda^{-1} \Lambda_{q_i} Q_\lambda^{-1}) = -\sum_{j=q_i}^{q_{i+1}-1} \phi_j'^T \phi'_j \,.$$

It is clear that

$$\frac{\partial \, \text{tr}(Q_\lambda^{-1})}{\partial \lambda_{u_i}} < 0 \,, \tag{D.5}$$

thus tr(Q_λ^{-1}) is monotone decreasing with respect to each λ_{u_i}. □

Part II

Non-Asymptotic Bounds on the MCMC Estimation Error

Chapter 1

Introduction

Let $\mathcal{X}$ be a Polish space (a separable, completely metrizable topological space) with Borel σ-algebra $\mathcal{B}(\mathcal{X})$ and let π be a probability measure. Many problems in Bayesian inference can be written as

$$\mathbb{E}_\pi(f) = \int_{\mathcal{X}} f(x)\pi(dx), \tag{1.1}$$

which is an intractable integral one wants to compute. Assume there is a Harris ergodic Markov Chain $\{X_m\}_{m=0}^{\infty}$ that converges to π. If one simulates m draws from this Markov chain, and $\mathbb{E}_\pi|f| < \infty$, then the following holds:

$$\hat{f}_m = \frac{1}{m}\sum_{i=1}^{m} f(X_i) \to \mathbb{E}_\pi(f) \quad \text{as } m \to \infty \ ,$$

with probability 1 by the strong law of large numbers. To assess the qualtity of this estimation, we define the root mean square error (RMSE) as

$$RMSE_x := \sqrt{\mathbb{E}_x\left((\hat{f}_m - \mathbb{E}_\pi(f))^2\right)}.$$

In a paper due to Latuszyński et al. (2013), the authors derive non-asymptotic bounds on the RMSE of estimates from Markov Chain Monte Carlo (MCMC) algorithms. In this paper, we use results from Latuszyński et al. (2013) to obtain finite, non-asymptotic bounds on the RMSE in several Bayesian statistical models.

There are three assumptions that must be established in order to use Latuszyński et al. (2013)'s results to bound the RMSE.

Drift Condition: There exist constants $0 \leq \lambda < 1, 0 < K < \infty$ and function $V : \mathcal{X} \to [1, \infty)$ s.t.

$$PV(x) := \mathbb{E}\left(V(X_{n+1})|X_n = x\right) \leq \begin{cases} \lambda V(x), & x \notin J \\ K, & x \in J. \end{cases} \tag{1.2}$$

Minorization Condition: There exist Borel set $J \subseteq \mathcal{X}$ of positive π measure, constant $0 < \delta < 1$, and probability measure ν such that for all $A \in \mathcal{B}(X)$,

$$P(x, A) \geq \delta\mathbb{I}(x \in J)\nu(A). \tag{1.3}$$

V**-Norm Condition:** For the function f from (1.1), define $\overline{f}(x) := f(x) - E_\pi(f)$. Then the following must hold:

$$||\overline{f}||_{V^{\frac{1}{2}}} := \sup_{x\in\mathcal{X}} \frac{|\overline{f}(x)|}{\sqrt{V(x)}} < \infty. \tag{1.4}$$

With these conditions established, we are now in a place to describe a way to upper bound the RMSE, which is given in Theorems 3.1, 4.2, and 4.5 in Latuszyński et al. (2013).

$$RMSE = \sqrt{\mathbb{E}_x\left(\overline{f}_m - E_g(f)\right)^2} \leq \frac{\sigma_{as}(P,f)}{\sqrt{m}}\left(1 + 2\frac{C_0(P,f)}{m}\right)^{\frac{1}{2}} + \frac{C_1(P,f)}{m} + \frac{C_2(P,f)}{m}, \tag{1.5}$$

where $\sigma_{as}^2(P,f), C_0(P,f), C_1(P,f)$, and $C_2(P,f)$ are constants that we can upper bound with the constants from Appendix A.

Steps to Upper-Bound RMSE

For a function f from (1.1), we must

1. Establish the Drift Condition for some function V and constants $\lambda < 1, K < \infty$.
2. Establish the Minorization Condition for some $\delta > 0$, set J and probability measure ν.
3. Establish the V-Norm Condition for f and the drift function V.
4. Using the bounds given in Appendix A, obtain values of the constants from (1.5) dependent on δ, J, V, λ, and K from steps 1, 2, and 3.
5. Compute the root mean square error from (1.5).

Before continuing, we make a remark on step 5 above. The theorem derived in Latuszyński et al. (2013) is

$$RMSE \leq \frac{\sigma_{as}}{\sqrt{m}}\left(1 + \frac{C_0(P,f)}{m}\right) + \frac{C_1(P,f)}{m} + \frac{C_2(P,f)}{m} \tag{1.6}$$

which is an upper bound on (1.5) by the Bernoulli inequality, which states

$$(1 + rx) \leq (1 + x)^r$$

for $x \in [-1, \infty), r \in \mathbb{Z}^+$. It should be noted that the difference between (1.5) and (1.6) is small for large m. This paper uses (1.6) for consistency but 1.5 gives better bounds for small m.

As noted in Latuszyński et al. (2013), The Markov Chain Central Limit Theorem (CLT) states that

$$\sqrt{m}(\hat{f}_m - E_\pi(f)) \xrightarrow{d} \mathcal{N}(0, \sigma^2_{as}(P,f)),$$

where $\sigma_{as}(P,f)$ is the asymptotic variance. It is easy to show that

$$\lim_{m\to\infty} m\mathbb{E}(\hat{f}_m - E_\pi(f))^2 = \sigma^2_{as}(P,f).$$

This shows that $\sigma^2_{as}(P,f)$ is asymptotically correct and therefore cannot be improved, as Latuszyński et al. (2013) states.

The outline of this paper is the following. In Chapter 2, we give results on a Bayesian one sample model. In Chapter 3, we give results on a Bayesian linear regression model. In the Appendices we provide inequalities to bound the RMSE given in Latuszyński et al. (2013), proofs of lemmas, and alternatives to the proofs in the body of the paper.

Chapter 2

The One-Sample Normal Model

2.1 Analysis of the μ-chain

We consider the one sample model with a normal prior distribution for μ and a Gamma prior distribution for τ. Assume $Y_1, Y_2, \ldots, Y_n|\mu, \tau \overset{iid}{\sim} \mathcal{N}(\mu, \frac{1}{\tau})$ with priors

$$\mu \sim \mathcal{N}\left(a, \frac{1}{b}\right) \quad \perp \quad \tau \sim \text{Gamma}(c, d).$$

Then after applying Bayes formula, we obtain the following conditional distributions:

$$\mu|\tau \sim \mathcal{N}(\hat{\mu}, \frac{1}{n\tau + b}), \quad \hat{\mu} = w\overline{y} + (1-w)a, \quad w = \frac{n\tau}{n\tau + b},$$

and

$$\tau|\mu \sim \text{Gamma}\left(c + \frac{n}{2}, d + \frac{(n-1)s^2 + (\mu - \overline{y})^2}{2}\right).$$

One can consider w as the weight between the frequentist estimate and the Bayesian estimate of $\hat{\mu}$. It is easy to see that as $n \to \infty$, $\hat{\mu} \to \overline{y}$, which says that for large sample sizes, the posterior mean is closer to the frequentist estimate. We now give two approaches that lead to different bounds on the RMSE. We are considering a Gibbs sampler that updates in the following way: $(\tau_m, \mu_m) \to (\tau_{m+1}, \mu_m) \to (\tau_{m+1}, \mu_{m+1})$. This creates a Markov Chain $\{(\tau_m, \mu_m)\}_{m=0}^{\infty}$. In the following section, we work with the μ-chain $\{\mu_m\}_{m=0}^{\infty}$ and we establish drift, minorization, and V-norm conditions that allow us to estimate the RMSE of functions of μ only. In Section 2.2, we establish the same conditions necessary to bound the RMSE of functions of τ.

Theorem 3. *The drift condition is satisfied using*

$$V(\mu_m) = (\mu_m - \overline{y})^2 + 1, \quad K = L + \rho\omega^2, \quad \lambda = \frac{L + \rho\omega^2}{\omega^2 + 1},$$

provided $\omega > \sqrt{\frac{L-1}{1-\rho}}$, *where* $\rho = \frac{1}{2c+n-2}$, $L = \rho\frac{2d+(n-1)s^2}{n} + (\overline{y} - a)^2 + 1$, *and* $J = [\overline{y} - \omega, \overline{y} + \omega]$.

Proof. Using the law of iterated expectations, we have

$$\begin{aligned} PV(\mu_{m+1}) &= \mathbb{E}[(\mu_m - \overline{y})^2 + 1|\mu_{m+1}] \\ &= \mathbb{E}[\mathbb{E}[(\mu_m - \overline{y})^2 + 1|\tau_{m+1}]|\mu_{m+1}] \\ &= \mathbb{E}[\text{Var}(\mu_m|\tau_{m+1}) + (\mathbb{E}[\mu_m - \overline{y}|\tau_{m+1}])^2\,|\mu_{m+1}] + 1. \end{aligned}$$

Then, since $\mu|\tau \sim \mathcal{N}(\hat{\mu}, \frac{1}{n\tau+b})$, with $\hat{\mu} = w\overline{y} + (1-w)a$, $w = \frac{n\tau}{n\tau+b}$, we have that

$$PV(\mu_{m+1}) = \mathbb{E}\left[\frac{1}{n\tau_{m+1} + b} + (1-w)^2(\overline{y} - a)^2|\mu_{m+1}\right] + 1,$$

which we simply bound above by

$$\frac{1}{n}\mathbb{E}\left[\frac{1}{\tau_{m+1}}|\mu_{m+1}\right] + (\overline{y}-a)^2 + 1.$$

Since $\tau|\mu \sim \text{Gamma}\left(c+\frac{n}{2}, d+\frac{(n-1)s^2+n(\mu-\overline{y})^2}{2}\right)$, we have

$$\begin{aligned} PV(\mu_{m+1}) &\leq \frac{1}{n}\left[\frac{2d+(n-1)s^2+n(\mu_{m+1}-\overline{y})^2}{2c+n-2}\right] + (\overline{y}-a)^2+1 \\ &= \frac{(\mu_{m+1}-\overline{y})^2}{2c+n-2} + \frac{1}{n}\left[\frac{2d+(n-1)s^2}{2c+n-2}\right] + (\overline{y}-a)^2+1 \\ &= \frac{(\mu_{m+1}-\overline{y})^2+1}{2c+n-2} + \frac{1}{n}\left[\frac{2d+(n-1)s^2}{2c+n-2}\right] + (\overline{y}-a)^2+1-\frac{1}{2c+n-2}. \end{aligned}$$

For convenience we define the constants $\rho := \frac{1}{2c+n-2}$ and $L := \rho\frac{2d+(n-1)s^2}{n} + (\overline{y}-a)^2+1$. Then the above bound on $PV(\mu_{m+1})$ can be written as

$$PV(\mu_{m+1}) \leq \rho V(\mu_m) + L - \rho.$$

For $\mu_m \in J := [\overline{y}-\omega, \overline{y}+\omega]$,

$$PV(\mu_{m+1}) \leq \rho V(\mu_{m+1}) + L - \rho \leq \rho(\omega^2+1) + L - \rho = \rho\omega^2 + L =: K.$$

For $\mu_m \notin J$, the drift condition requires $L-\rho \leq (\lambda-\rho)V(\mu_m)$ for some constant $0<\lambda<1$. By setting

$$\lambda := \frac{K}{\omega^2+1} = \frac{L+\rho\omega^2}{\inf\limits_{\mu_m\notin J} V(\mu_m)} = \sup_{\mu_m\notin J}\frac{L-\rho}{V(\mu_m)} + \rho \geq \frac{L-\rho}{V(\mu_m)} + \rho,$$

the drift condition is established. Lastly, since we require $\lambda < 1$, it must be true that

$$\omega > \sqrt{K-1}.$$

□

We now prove the minorization condition for the μ-chain. The transition density function is

$$p(\mu_{m+1}|\mu_m) = \int_{\mathbb{R}^+} p(\mu_{m+1}|\tau)p(\tau|\mu_m)\,d\tau.$$

We can easily see that

$$p(\mu_{m+1}|\mu_m) \geq \int_{\mathbb{R}} p(\mu_{m+1}|\tau)\inf_{\mu_m\in J} p(\tau|\mu_m)\,d\tau.$$

Then, using a calculation similar to Jones and Hobert (2001),

$$g(\tau) := \inf_{\mu\in J} p(\tau|\mu) \leq \begin{cases} \text{Gamma}\left(c+\frac{n}{2}, d+\frac{(n-1)s^2}{2}\right) & \tau \leqslant \tau^* \\ \text{Gamma}\left(c+\frac{n}{2}, d+\frac{(n-1)s^2+n\omega^2}{2}\right) & \tau > \tau^*, \end{cases}$$

where $\frac{2c+n}{n\omega^2}\log\left(1+\frac{n\omega^2}{2d+(n-1)s^2}\right) := \tau^*$. Then the minorization condition is satisfied if we use

$$\delta := \int_{\mathbb{R}^+}\int_{\mathbb{R}} p(\mu|\tau)g(\tau)\,d\mu\,d\tau = \int_{\mathbb{R}^+} g(\tau)\,d\tau,$$

by Fubini's Theorem. Note that the minorization condition is independent of the drift function.
It is enough to show that the norm of f is finite, where $f(\mu) = (\mu-\overline{y})$.

$$||f||_{V^{1/2}} = \sup_{\mu\in\mathbb{R}}\frac{|\mu-\overline{y}|}{\sqrt{(\mu-\overline{y})^2+1}} < \infty$$

Note that in the above computations, we simplified our work by defining $f(\mu) = (\mu-\overline{y})^j$. The values of the RMSE one obtains when using this value of $||f||_{V^{1/2}}$ are for the random variable $(\mu-\overline{y})^j$. Shifting by $\overline{y}$ does not change the RMSE, since the RMSE is invariant under shifts.

One is often interested in finding the minimum m required such that

$$\mathbb{P}\left(|\hat{f}_m - E_\pi(f)| \leq \epsilon\right) > 1-\alpha,$$

for some $\epsilon > 0,\ \alpha \in (0,1)$. This is equivalent to finding the minimum m such that

$$RMSE(\hat{f}_m) \leq \epsilon\sqrt{\alpha} \tag{2.1}$$

by Chebyshev's inequality. In Table 2.1 we fix $\alpha = .05$ and give the number of iterations required to bound (2.1) for different ϵ. We are using the Diasorin dataset from chapter 5 of Christensen et al. (2010).

Table 2.1: Bounds on the RMSE for the posterior mean of μ, Required Number of Iterations m.

ϵ	RMSE Bound	m
.25	0.0559017	4000
.125	0.02795085	15000
.01	0.002236068	2190000
.005	0.001118034	8720000

2.2 Analysis of the Gibbs Chain

In this section we prove that it is possible to prove the drift condition, minorization condition, and V-norm condition for drift functions of the form $V(\tau,\mu) = (\mu-\overline{y})^2 + \tau^r + \tau^{-s} + \eta$, for $\eta < 1$. The advantage of using the joint drift function is that the minorization condition allows the user to compute means of functions of both variables, while only having to establish the minorization condition for one of the variables. In the one sample model with normal prior distribution, the minorization conditions are easy, but in the regression case, the minorization condition for the τ-chain is intractable, so we must resort to a joint drift function in this case.

2.2.1 Drift Condition

Theorem 4. *For the drift function $V(\tau_m,\mu_m) = (\mu_m-\overline{y})^2 + \tau_m^r + \tau_m^{-s} + \eta$ with $\eta := 1 - \left(\frac{s}{r}\right)^{\frac{r}{r+s}} - \left(\frac{r}{s}\right)^{\frac{s}{r+s}}$,*

$$PV(\tau_{m+1},\mu_{m+1}) \leq \begin{cases} K & \text{if } (\tau_m,\mu_m) \in J \\ \lambda V(\tau_m,\mu_m) & \text{if } (\tau_m,\mu_m) \notin J, \end{cases}$$

where $J := \{(\tau,\mu) \in \mathbb{R}^+ \times \mathbb{R} : (\mu-\overline{y})^2 \leq \omega_1^2,$ and $\tau^r + \tau^{-s} \leq \omega_2\}$, $K := \rho(\omega_1^2+\omega_2^2+\eta) + L$, and $\lambda := \frac{L}{\omega_1^2+\omega_2^2+\eta} + \rho$.

Proof. By the construction of η, the range of $V(\tau_m,\mu_m)$ is now $[1,\infty)$. We begin by splitting PV into three main expectations. We decide to bound $\frac{1}{n\tau_m+b}$ by $\frac{1}{n\tau_m}$ since this is a better bound.

$$\begin{aligned} PV(\tau_{m+1},\mu_{m+1}) &= \mathbb{E}\left((\mu_{m+1}-\overline{y})^2 + \tau_{m+1}^r + \tau_{m+1}^{-s} + \eta|\mu_m,\tau_m\right) \\ &= \mathbb{E}\left(\frac{1}{n\tau_m+b} + (1-w)^2(\overline{y}-a)^2|\mu_m\right) + \mathbb{E}(\tau_m^r|\mu_m) + \mathbb{E}(\tau_m^{-s}|\mu_m) + \eta. \end{aligned}$$

This comes from the definition of the second moment of the normally distributed $\mu_m|\tau_m$. We then evaluate the two rightmost expectations and split the left further into two pieces. We also define $\zeta := 2d + (n-1)s^2$ for convenience.

$$\begin{aligned} PV(\tau_{m+1},\mu_{m+1}) \leq{}& \frac{1}{n}\mathbb{E}\left(\frac{1}{\tau_m}|\mu_m\right) + b^2(\overline{y}-a)^2\mathbb{E}\left(\left(\frac{1}{n\tau_m+b}\right)^2|\mu_m\right) + D_1(\zeta+(\mu_m-\overline{y})^2)^{-r} \\ &+ D_2(\zeta+(\mu_m-\overline{y})^2)^s + \eta. \end{aligned}$$

By expanding the denominator of the expectation of the square of the variance of μ, we upper bound this quantity by dropping each term besides the middle $2n\tau_m$ term. After rewriting, we combine like terms to arrive at

$$PV(\tau_{m+1},\mu_{m+1}) \leq \left(\frac{2+b(\overline{y}-a)^2}{2n}\right)\frac{\zeta+(\mu_m-\overline{y})^2}{2c+n-2} + D_1\zeta^{-r} + D_2\zeta^s + D_2(\mu_m-\overline{y})^{2s} + \eta.$$

Here we have also used the fact that $s \in (0,1)$ to bound the right hand μ_m-dependent term by the function of μ_m shown. This function of μ_m is bounded above by the same function shifted up by 1, and we have

$$\begin{aligned}PV(\tau_{m+1},\mu_{m+1}) &\leq \left(\frac{2+b(\overline{y}-a)^2}{2n(2c+n-2)}\right)\left[\zeta+(\mu_m-\overline{y})^2\right] + D_1\zeta^{-r} + D_2\left[(\mu_m-\overline{y})^2+1\right] + \eta \\ &= \zeta\frac{2+b(\overline{y}-a)^2}{2n(2c+n-2)} + (\mu_m-\overline{y})^2\frac{2+b(\overline{y}-a)^2}{2n(2c+n-2)} + D_1\zeta^{-r} + D_2\zeta^s \\ &+ D_2\left[(\mu_m-\overline{y})^2+1\right] + \eta.\end{aligned}$$

Now we define $\rho := D_2 + \frac{2+b(\overline{y}-a)^2}{2n(2c+n-2)}$ and by some algebraic manipulations recover the drift function:

$$\begin{aligned}PV(\tau_{m+1},\mu_{m+1}) &\leq \zeta(\rho-D_2) + \rho\left[(\mu_m-\overline{y})^2+1\right] + D_2 - \rho + D_1\zeta^{-r} + D_2\zeta^s + \eta \\ &\leq \rho V(\tau_m,\mu_m) + D_2 - \rho + \zeta(\rho-D_2) + D_1\zeta^{-r} + D_2\zeta^s + \eta \\ &= \rho V(\tau_m,\mu_m) + D_2(1-\zeta+\zeta^s) + \rho(\zeta-1) + D_1\zeta^{-r} + \eta.\end{aligned}$$

Now we define the constant

$$L := D_2(1-\zeta+\zeta^s) + \rho(\zeta-1) + D_1\zeta^{-r} + \eta.$$

Then we rewrite our bound as

$$PV(\tau_{m+1},\mu_{m+1}) \leq \rho V(\tau_m,\mu_m) + L.$$

For $(\tau_m,\mu_m) \in J$, we bound above by

$$PV(\tau_{m+1},\mu_{m+1}) \leq \rho(\omega_1^2+\omega_2^2+\eta) + L =: K.$$

For $(\tau_m,\mu_m) \notin J$, we require

$$PV(\tau_{m+1},\mu_{m+1}) \leq \rho V(\tau_m,\mu_m) + L \leq \lambda V(\tau_m,\mu_m),$$

which yields

$$\lambda := \frac{L}{\omega_1^2+\omega_2^2+\eta} + \rho \geq \frac{L}{V(\tau_m,\mu_m)} + \rho,$$

for all $(\tau_m,\mu_m) \notin J$. Since we require $\lambda < 1$, we have that

$$\omega_1^2+\omega_2^2 > \frac{L}{1-\rho} - \eta.$$

□

2.2.2 Minorization Condition

We now turn to proving the minorization condition for this chain. First we consider the transition kernel, defined as follows:

$$\begin{aligned}f((\tau_{m+1},\mu_{m+1})|(\tau_m,\mu_m)) &= f_{\mu_{m+1}|\tau_{m+1}}(\mu_{m+1}|\tau_{m+1})f_{\tau_{m+1}|\mu_m}(\tau_{m+1}|\mu_m) \\ &\geq f_{\mu_{m+1}|\tau_m}(\mu_{m+1}|\tau_{m+1}) \inf_{(\tau_m,\mu_m)\in J^\star} f_{\tau_{m+1}|\mu_m}(\tau_{m+1}|\mu_m),\end{aligned}$$

where $J^\star := \{(\tau,\mu) \in \mathbb{R}^+ \times \mathbb{R} : (\mu-\overline{y})^2 \leq \omega_1\}$, and we know that $\inf J^\star \geq \inf J$. Note also that this is the infimum over all $(\tau_m,\mu_m) \in J^\star$. Then we recall the following theorem.

$$g(\tau_{m+1}) := \inf_{(\tau_m,\mu_m)\in J^\star} p(\tau_{m+1}|\mu_m) \leq \begin{cases}\text{Gamma}\left(c+\frac{n}{2}, d+\frac{(n-1)s^2}{2}\right) & \tau_{m+1} \leqslant \tau* \\ \text{Gamma}\left(c+\frac{n}{2}, d+\frac{(n-1)s^2+n\omega_1^2}{2}\right) & \tau_{m+1} > \tau^*,\end{cases}$$

where $\tau^\star = \frac{2c+n}{n\omega_1^2}\log\left(1+\frac{n\omega_1^2}{\zeta}\right)$. Then the minorization condition is satisfied if we use

$$\delta := \int_{\mathbb{R}^+}\int_{\mathbb{R}} p(\mu|\tau)g(\tau)\,d\mu\,d\tau = \int_{\mathbb{R}} g(\tau)\,d\tau.$$

2.2.3 V-norm Condition

We now prove that the V-norm conditions established in preceding sections may be used to establish the V-norm condition in the joint drift case. Consider $f(\mu) = \mu - \overline{y}$. Then,

$$\begin{aligned} ||f||_{V^{1/2}} &= \sup_{(\tau,\mu)\in\mathbb{R}^+\times\mathbb{R}} \frac{|\mu - \overline{y}|}{\sqrt{(\mu-\overline{y})^2 + \tau^r + \tau^{-s} + \eta}} \\ &\leq \sup_{(\tau,\mu)\in\mathbb{R}^+\times\mathbb{R}} \frac{|\mu - \overline{y}|}{\sqrt{(\mu-\overline{y})^2}}. \\ &= \sup_{\mu\in\mathbb{R}} \frac{|\mu - \overline{y}|}{\sqrt{(\mu-\overline{y})^2}} \end{aligned}$$

Then one can simply use results from the V-norm condition on the μ-chain to establish the minorization condition. A comparable approach can be used to establish the V-norm condition for functions of τ.

Chapter 3

The Linear Regression Model

3.1 Introduction

Suppose we have a vector of data, Y, such that $Y|\beta, \tau \sim \mathcal{N}_n(X\beta, \frac{1}{\tau}I_n)$ with the following priors:

$$\beta \sim \mathcal{N}_p(\beta_0, C_0) \quad \perp \quad \tau \sim \text{Gamma}(a, b).$$

After applying Bayes formula, we obtain the following posterior conditional distributions:

$$\tau|\beta, Y \sim \text{Gamma}\left(\frac{n}{2} + a, b + \frac{\hat{\sigma}^2(n-p) + ||X\beta - X\hat{\beta}||^2}{2}\right)$$

$$\beta|\tau, Y \sim \mathcal{N}_p(m, \Phi),$$

where $\Phi := (\tau X^T X + C_0^{-1})^{-1}, m := \Phi\left[\tau X^T Y + C_0^{-1}\beta_0\right], \hat{\sigma}^2 := \frac{||Y - X\hat{\beta}||^2}{n-p}$, and $\hat{\beta} := (X^T X)^{-1} X^T Y$. We first present some linear algebra techniques that prove useful when establishing the drift condition. Let A be an $n \times n$ symmetric matrix. We say that A is non-negative definite (nnd) if for all $x \in \mathbb{R}^n$,

$$x^T A x \geq 0.$$

We say that A is positive definite (pd) if for all $x \in \mathbb{R}^n$,

$$x^T A x > 0.$$

It is easy to see that the sum of a non-negative definite matrix and a positive definite matrix is a positive definite matrix (similar conclusions can be reached for the sum of positive definite matrices and the sum of non-negative definite matrices). It is also easy to see that positive definite matrices are invertible. For symmetric $n \times n$ matrices A, B, we now define a relation $\preceq$ by

$$A \preceq B \quad \text{iff} \quad A - B \text{ is a non-negative definite matrix.}$$

It is easy to see that if $A \preceq B$, then $tr(A) \leq tr(B)$. To prove this, pick an approprite $x \in \mathbb{R}^n$ and use the definition of $\preceq$. We also use the fact that the trace operator is cyclic, meaning that

$$tr(AB) = tr(BA),$$

for matrices A, B of appropriate size. One can also prove that $A \preceq B$ if and only if $A^{-1} \succeq B^{-1}$. Finally, we remind the reader that covariance matrices are always non-negative definite. We use the norm $||A|| = \sqrt{(}tr(A^T A))$. With these tools, we are now in a position to establish the drift condition, minorization condition, and V-norm condition for the regression model.

3.2 Analysis of the β-chain

Theorem 5. *There exist constants $\lambda \in (0,1)$ and $K \in (0, \infty)$, set $J \subseteq \mathbb{R}^p$ such that for all $\beta \in \mathbb{R}^p$,*

$$PV(\beta_{m+1}) := \mathbb{E}\left(V(\beta_{m+1})|\beta_m = \beta\right) \leq \begin{cases} K & \text{if } \beta \in J \\ \lambda V(\beta) & \text{if } \beta \notin J, \end{cases}$$

provided $\omega > \sqrt{\frac{L-1}{1-\rho}}$, where $V(\beta) := ||X\beta - X\hat{\beta}||^2 + 1, K = L + \rho\omega^2, \rho = \frac{\gamma_1}{n+2a-2}, \lambda = \frac{L+\rho\omega^2}{\omega^2+1}$.

Proof. We begin by using the law of iterated expectations:

$$PV(\beta_{m+1}) = \mathbb{E}\left[\mathbb{E}\left(||X(\beta_{m+1}-\hat{\beta})||^2|\tau_m\right)|\beta_m\right).$$

We focus first on the inner expectation.

$$\mathbb{E}\left(||X(\beta_{m+1}-\hat{\beta})||^2|\tau_m\right) = 1 + \text{tr}(X^TX\Phi) + \mathbb{E}(\beta_{m+1}-\hat{\beta}|\tau_m)X^TX\mathbb{E}(\beta_{m+1}-\hat{\beta}|\tau_m) \leq \frac{\gamma_1}{\tau} + \gamma_2$$

for constants $\gamma_1, \gamma_2 \in \mathbb{R}$ found numerically to be $\gamma_1 \approx 2, \gamma_2 \approx 2 \times 10^{-15}$. Then, we obtain

$$PV(\beta_{m+1}) \leq 1 + \gamma_2 + \gamma_1 \frac{(2b + \hat{\sigma}^2(n-p) + ||X(\beta_m - \hat{\beta})||^2)}{n+2a-2}.$$

We can rewrite the above equation as

$$\begin{aligned} PV(\beta_{m+1}) &\leq 1 + \gamma_2 - \frac{\gamma_1}{n+2a-2} + \gamma_1\frac{(2b+\hat{\sigma}^2(n-p))}{n+2a-2} + \frac{\gamma_1}{n+2a-2}(||X(\beta_m-\hat{\beta})||^2+1). \\ &= 1 + \gamma_2 - \frac{\gamma_1}{n+2a-2} + \gamma_1\frac{(2b+\hat{\sigma}^2(n-p))}{n+2a-2} + \frac{\gamma_1}{n+2a-2}V(\beta_m). \\ &= L - \rho + \rho V(\beta_m), \end{aligned}$$

where $\rho = \frac{\gamma_1}{n+2a-2}$ and $L = 1 + \gamma_2 + \rho(2b + \hat{\sigma}^2(n-p))$. Then, for $\beta_m \in J$, we have that $PV(\beta_{m+1}) \leq K$, where $K := L + \rho\omega^2$. For $\beta_m \notin J$, we have that $\lambda = \frac{L+\rho\omega^2}{\omega^2+1}$. In order for $\lambda < 1$, we require

$$\frac{L-\rho}{\omega^2+1} + \rho < 1 \quad \text{iff} \quad \omega > \sqrt{\frac{L-1}{1-\rho}}.$$

□

We give two examples to justify our definition of λ.

Example 1: Consider $X^TX = \mathbb{I}_p$, with $p = 2$. Then, $V(\beta) = \beta_1^2 + \beta_2^2 + 1$. In order to guarantee that

$$PV(\beta_{m+1}) \leq \rho V(\beta_m) + L - \rho \leq \lambda V(\beta_m)$$

for $\beta \notin J$, we want to pick λ such that $\lambda = K/V_{min}$, where V_{min} is the minimum value of V outside J. From Figure 3.1, we see that V_{min} occurs along the boundary of J. In other words, V_{min} occurs when $||X(\beta-\hat{\beta})||^2 = \omega^2$. To summarize our conclusion, what we have is the following optimization problem:

$$\begin{aligned} &\underset{\beta\in\mathbb{R}^2/J}{\text{minimize}} && ||X(\beta-\hat{\beta})||^2 + 1, \\ &\text{subject to the constraint} && ||X(\beta-\hat{\beta})||^2 \geq \omega^2. \end{aligned}$$

Example 2: Consider

$$X = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \textit{ such that } X^TX = \begin{pmatrix} 10 & 14 \\ 14 & 20 \end{pmatrix}$$

with $p = 2$. Then, $V(\beta) = 10\beta_1^2 + 28\beta_1\beta_2 + 20\beta_2^2 + 1$, which is shown in Figure 3.2. Although V no longer has the same shape as it did in Example 1, we can still set $\lambda = K/V_{min}$, where $V_{min} = \omega^2 + 1$, which occurs along ∂J. The same approach can be applied for all matrices X and all values of p. The only change in this case is the shape of the set J.

3.2.1 Minorization Condition

In order to prove the minorization condition for the β-chain of the regression case, we refer to the following proposition, which lower bounds the transition density function.

$$\begin{aligned} p(\beta_{m+1}|\beta_m) &= \int_{\mathbb{R}^+} p(\beta_{m+1}|\tau)p(\tau|\beta_m)\, d\tau. \\ &\geq \int_{\mathbb{R}^+} p(\beta_{m+1}|\tau) \inf_{\beta_m\in J} p(\tau|\beta_m)\, d\tau. \end{aligned}$$

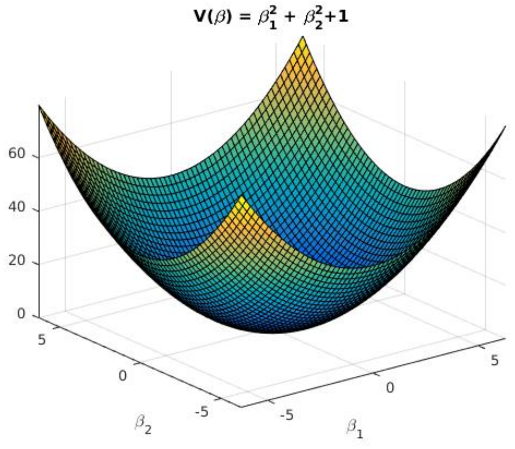

Figure 3.1: $V(\beta)$, Example 1

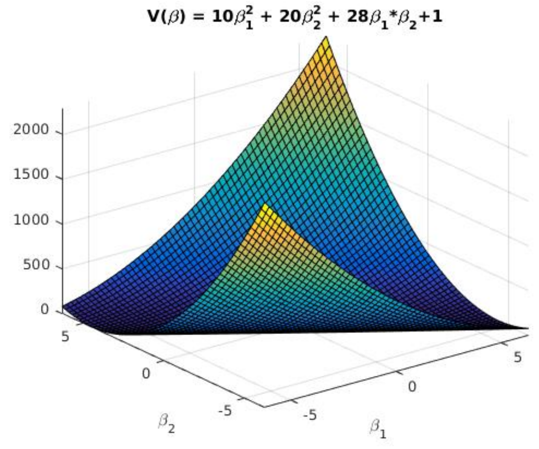

Figure 3.2: $V(\beta)$, Example 2

Now, using calculations similar to the ones in Jones and Hobert (2001), we know that for $g(\tau_m) := \inf_{\beta_m \in J} p(\tau_m|\beta_m)$, where $J \subseteq \mathbb{R}^p$ is the small set discussed previously,

$$g(\tau_m) = \begin{cases} \Gamma(a + \frac{n}{2}, b + \frac{(n-p)\sigma^2}{2}) & \tau_m \leq \tau^\star \\ \Gamma(a + \frac{n}{2}, b + \frac{(n-p)\sigma^2 + \omega^2}{2}), & \tau_m > \tau^\star. \end{cases}$$

where $\tau^\star = \frac{2a+n}{\omega^2} \log\left(1 + \frac{\omega^2}{2b+(n-p)\sigma^2}\right)$. Then if we define

$$\delta := \int_{\mathbb{R}^p} \int_{\mathbb{R}^+} p(\beta|\tau) g(\tau) d\tau d\beta = \int_{\mathbb{R}^+} g(\tau) d\tau,$$

the simplification of which is due to Fubini's theorem, the minorization condition is satisfied for the β-chain if we define $\nu(\beta) := \delta^{-1} \int_{\mathbb{R}^+} p(\beta|\tau) g(\tau)\, d\tau$. This integral can be computed using the incomplete gamma function.

3.2.2 V-norm Condition

In this section we discuss the conditions necessary to establish the V-norm condition. We are going to use the following fact:

$$\lambda_{min} \mathbb{I}_n \preceq X^T X \preceq \lambda_{max} \mathbb{I}$$

where $\lambda_{min}, \lambda_{max}$ denote respecively the minimum and maximum eigenvalue of $X^T X$ that one can obtain easily by the spectral decomposition of $X^T X$. Then,

$$||f||_{V^{1/2}} = \sqrt{\lambda_{min}}.$$

One area of future work would be to try to improve this bound. For the log NBA data file, we find that $||f||_{V^{1/2}} \approx 8.9$. (See the Example in Section 3.3.4.)

3.3 Analysis of the Gibbs Chain

3.3.1 Drift Condition

We now extend the results of the methods above to the (τ_m, β_m)-chain in the regression model using the $\tau_m \to \beta_m \to \tau_{m+1} \to \beta_{m+1}$ chain using the drift function $V : \mathbb{R}^+ \times \mathbb{R}^p \to [1, \infty)$ by $V(\tau_m, \beta_m) := ||X\beta_m - X\hat{\beta}||^2 + \tau_m^2 + \tau_m^{-1} + \eta$, where $1 > \eta := 1 - \left(\frac{s}{r}\right)^{\frac{r}{r+s}} - \left(\frac{r}{s}\right)^{\frac{s}{r+s}}$.

Theorem 6. *There exist constants $\lambda \in (0,1)$ and $K \in \mathbb{R}^+$ and a set $J \subseteq \mathbb{R} + \times \mathbb{R}^p$ such that for every $(\tau, \beta) \in \mathbb{R}^+ \times \mathbb{R}^p$,*

$$PV(\tau, \beta) := \mathbb{E}\left(V(\tau_{m+1}, \beta_{m+1}) | (\tau_m, \beta_m) = (\tau, \beta)\right) \leq \begin{cases} K & \textit{if } (\tau, \beta) \in J \\ \lambda V(\tau, \beta) & \textit{if } (\tau, \beta) \notin J, \end{cases}$$

where $V(\tau,\beta) := ||X(\beta-\hat{\beta})||^2 + \tau^2 + \frac{1}{\tau} + \eta$, $\eta := 1 - \left(\frac{s}{r}\right)^{\frac{r}{r+s}} - \left(\frac{r}{s}\right)^{\frac{s}{r+s}}$, $L = 1 + \gamma_2 + \frac{D_1}{\zeta^2} + \zeta(A + D_2)$, $\rho = A + D_2$, $K = \rho(\omega_1^2 + \omega_2)$, *and* $J = \{(\tau,\beta) \in \mathbb{R}^+ \times \mathbb{R}^P : ||X(\beta-\hat{\beta})||^2 \le \omega_1^2, \tau^2 + \frac{1}{\tau} \le \omega_2\}$.

Proof. We begin by using the law of iterated expectations:

$$\begin{aligned}PV(\tau_{m+1},\beta_{m+1}) &= \mathbb{E}\left[\mathbb{E}\left(||X(\beta_{m+1}-\hat{\beta})||^2 + \tau_{m+1}^2 + \tau_{m+1}^{-1})|\tau_{m+1}\right)|\beta_m,\tau_m\right)\\ &= \mathbb{E}\left[\mathbb{E}\left(||X(\beta_{m+1}-\hat{\beta})||^2|\tau_{m+1}\right) + \tau_{m+1}^2 + \tau_{m+1}^{-1})|\beta_m\right)\\ &\le 1 + \gamma_2 + (p+\gamma_1)\frac{(2b + \hat{\sigma}^2(n-p) + ||X(\beta_m - \hat{\beta})||^2)}{n+2a-2} + \mathbb{E}\left(\tau_{m+1}^2|\beta_m\right) + \mathbb{E}\left(\tau_{m+1}^{-1}|\beta_m\right).\end{aligned}$$

This comes from our earlier work in the β_m-chain, where we bounded the β_m-dependent portion of the expectation using

$$\mathbb{E}\left(||X(\beta_{m+1}-\hat{\beta})||^2|\tau_{m+1}\right) = 1 + \text{tr}(X^TX\Phi) + \mathbb{E}(\beta_{m+1}-\hat{\beta}|\tau_{m+1})X^TX\mathbb{E}(\beta_{m+1}-\hat{\beta}|\tau_{m+1}) \le \frac{\gamma_1}{\tau} + \gamma_2,$$

for constants $\gamma_1, \gamma_2 \in \mathbb{R}$. We've also seen the τ_m portion of the PV bound before, and we bound it in the same way, namely:

$$\mathbb{E}\left(\tau_{m+1}^2|\beta_m\right) + \mathbb{E}\left(\tau_{m+1}^{-1}|\beta_m\right) \le D_1\left(\zeta + ||\beta_m - \hat{\beta}||^2\right)^{-2} + D_2\left(\zeta + ||\beta_m - \hat{\beta}||^2\right),$$

where $D_1 := \frac{4\Gamma(2+\alpha)}{\Gamma(\alpha)}$, $D_2 := \frac{\Gamma(\alpha-1)}{2\Gamma(\alpha)}$, $A := \frac{\gamma_1}{n+2a-2}$. Then we combine these previous results to arrive at the following upper bound:

$$\begin{aligned}PV(\tau_{m+1},\beta_{m+1}) &\le 1 + \gamma_2 + A(\zeta + ||X(\beta_m-\hat{\beta})||^2) + D_1\left(\zeta + ||X(\beta_m-\hat{\beta})||^2\right)^{-2} + D_2\left(\zeta + ||X(\beta_m-\hat{\beta})||^2\right)\\ &\le 1 + \gamma_2 + D_1(\zeta)^{-2} + (A+D_2)\zeta + (A+D_2)(||X(\beta-\hat{\beta}||^2 + \tau^2 + \tau^{-1})) = L + \rho V(\tau,\beta),\end{aligned}$$

where $L = 1 + \gamma_2 + D_1(\zeta)^{-2} + (A+D_2)\zeta, \rho = A + D_2$. For $(\tau_m,\beta_m) \in J = \{(\tau,\beta) \subseteq \mathbb{R}^+ \times \mathbb{R}^p : ||X(\beta-\hat{\beta})||^2 \le \omega_1^2, \tau^2 + \tau^{-1} \le \omega_2.\}$, we can bound above by the constant K as follows:

$$PV(\tau_{m+1},\beta_{m+1}) \le L + \rho(\omega_1^2 + \omega_2) =: K.$$

For $(\tau_m,\beta_m) \notin J$, we bound above

$$PV(\tau_{m+1},\beta_{m+1}) \le L + \rho V(\tau_m,\beta_m) \le \lambda V(\tau_m,\beta_m),$$

and solve this right hand inequality for λ. Thus,

$$\lambda := \frac{L}{\omega_1^2 + \omega_2} + \rho \ge \frac{L}{V(\mu_m,\tau_m)} + \rho.$$

Since we require $\lambda < 1$, we require

$$\omega_1^2 + \omega_2 \ge \frac{L}{1-\rho}. \tag{3.1}$$

□

Note that because of the definition of the set J, we have the ability to choose (ω_1,ω_2) according to (3.1). In our code we use the package *nloptr*[1], which is used for nonlinear optimization according to inequality or inequality constraints.

3.3.2 Minorization Condition for the Gibbs Chain

The advantage of using the joint drift function $V(\tau,\beta)$ is that the minorization condition for this chain is the same as that for the β-chain, which is given in Section 3.2.1.

[1]Available at: `https://cran.r-project.org/web/packages/nloptr/nloptr.pdf`

3.3.3 V-norm Condition

We now prove that the V-norm conditions established in preceding sections may be used to establish the V-norm condition in the joint drift regression case. Consider $f(\beta) = \beta_i$. Then,

$$\begin{aligned}||f||_{V^{1/2}} &= \sup_{(\tau,\beta)\in\mathbb{R}^+\times\mathbb{R}^p} \frac{|\beta_i|}{\sqrt{||X(\beta-\hat{\beta})||^2 + \tau^r + \tau^{-s} + \eta}} \\ &\leq \sup_{(\tau,\beta)\in\mathbb{R}^+\times\mathbb{R}^p} \frac{|\beta_i|}{\sqrt{||X(\beta-\hat{\beta})||^2}}. \\ &= \sup_{\beta\in\mathbb{R}^p} \frac{|\beta_i|}{||X(\beta-\hat{\beta})||^2}.\end{aligned}$$

Then one can simply use results from the V-norm condition on the β-chain to establish the minorization condition. A comparable approach can be used to establish the V-norm condition for functions of τ.

3.3.4 Example: Bounds on the RMSE

One is often interested in finding the mimimum m required such that

$$\mathbb{P}\left(|\hat{f}_m - E_g(f)| \leq \epsilon\right) > 1-\alpha \tag{3.2}$$

for some $\epsilon > 0,\ \alpha \in (0,1)$. This is equivalent to finding the minimimum m such that

$$RMSE(\hat{f}_m) \leq \epsilon\sqrt{\alpha} \tag{3.3}$$

by Chebyshev's inequality. We now fix $\alpha = .05$. In Table 3.1 we compute the number of iterations required to bound the value from (3.3) for different ϵ. When using $f(\beta) = \beta_i$, we use $||f||_{V^{1/2}} \approx 8.9$ and when using $f(\tau) = \tau$, we use $||f||_{V^{1/2}} \approx .63$, with $s = \frac{1}{2}, r = \frac{5}{2}$.

Table 3.1: RMSE bounds on β_i, $i = 1, 2$, Required Number of Iterations

ϵ	RMSE Bound	Theorem 5, β_i	Theorem 6, β_i	Theorem 6, τ
.25	0.0559017	80000	200000	1000
.125	0.02795085	3100000	800000	4200
.01	0.002236068	47800000	121800000	610000
.005	0.001118034	190900000	480000000	2440000

Table 3.1 shows that Theorem 5 requires fewer iterations of the Gibbs sampler, but Theorem 6 allows for estimation for functions of τ.

3.4 Conclusion

In this paper we have used results from Latuszyński et al. (2013) to derive non-asymptotic bounds on the RMSE of the usual MCMC estimators. We showed that it is possible to establish a drift condition, a minorization condition, and a V-norm condition for the one sample model with normal priors and the regression model. In each case we showed that these conditions can be established in various ways, highlighting optimal ways of proving such conditions in the future for more complicated models. We also worked with the case where one considers a joint drift function, which allows the user to avoid potentially intractable infimum computations in order to establish the minorization condition. We present numerical results to establish the accuracy of the theorems presented in this paper. Future areas of research include finding lower bounds on the V-norm in the regression model, bounding terms of the form

$$\mathbb{E}\left(\frac{1}{\mu+c}|\tau\right), \quad \text{where} \quad \mu|\tau \sim \mathcal{N}(a,b), \quad a,b,c \in \mathbb{R},$$

by some function of τ whose mean is easily computed, and extending the results of this paper to the linear mixed model and other more difficult statistical models.

Appendix A

General RMSE Bounds

Here we give upper bounds on the constants that appear in (1.5). See Latuszyński et al. (2013).

$$(i) \qquad C_0(P) \leq \frac{\lambda}{1-\lambda}\pi(V) + \frac{K-\lambda-\delta}{\delta(1-\lambda)} + \frac{1}{2},$$

$$(ii) \quad \frac{\sigma_{as}^2}{||\overline{f}||_{V^{1/2}}^2} \leq \frac{1+\sqrt{\lambda}}{1-\sqrt{\lambda}}\pi(V) + \frac{2(\sqrt{K}-\sqrt{\lambda}-\delta)}{\delta(1-\sqrt{\lambda})}\pi(V^{1/2}),$$

$$(iii) \quad \frac{C_1(P,f)^2}{||\overline{f}||_{V^{1/2}}^2} \leq \frac{1}{(1-\sqrt{\lambda})^2}\xi(V) + \frac{2(\sqrt{K}-\sqrt{\lambda}-\delta)^2}{\delta(1-\sqrt{\lambda})^2} + \frac{\delta(K-\lambda-\delta)+2(K^{1/2}-\lambda^{1/2}-\delta)^2}{\delta^2(1-\lambda^{1/2})^2},$$

$$(iv) \quad \frac{C_2(P,f)^2}{||\overline{f}||_{V^{1/2}}^2} \leq \frac{1}{(1-\sqrt{\lambda})^2}\xi P^n(V) + \frac{2(\sqrt{K}-\sqrt{\lambda}-\delta)^2}{\delta(1-\sqrt{\lambda})^2} + \frac{\delta(K-\lambda-\delta)+2(K^{1/2}-\lambda^{1/2}-\delta)^2}{\delta^2(1-\lambda^{1/2})^2}.$$

And more upper bounds on constants useful to calculating the above bounds:

$$(i) \quad \pi(V^{1/2}) \leq \pi(J)\frac{K^{1/2}-\lambda^{1/2}}{1-\lambda^{1/2}} \leq \frac{K^{1/2}-\lambda^{1/2}}{1-\lambda^{1/2}},$$

$$(ii) \qquad \pi(V) \leq \pi(J)\frac{K-\lambda}{1-\lambda} \leq \frac{K-\lambda}{1-\lambda},$$

$$(iii) \ \textit{if}\, \xi(V^{1/2}) \leq \frac{K^{1/2}}{1-\lambda^{1/2}} \ \textit{then} \ \xi P^n(V^{1/2}) \leq \frac{K^{1/2}}{1-\lambda^{1/2}},$$

$$(iv) \quad \textit{if}\, \xi(V) \leq \frac{K}{1-\lambda} \ \textit{then} \ \xi P^n(V) \leq \frac{K}{1-\lambda},$$

(v) $\quad ||\overline{f}||_{V^{1/2}}$ *can be related to* $||f||_{V^{1/2}}$ *by*

$$||\overline{f}||_{V^{1/2}} \leq ||f||_{V^{1/2}}\left[1+\frac{\pi(J)(K^{1/2}-\lambda^{1/2})}{(1-\lambda^{1/2})\inf_{x\in\mathcal{X}} V^{1/2}(x)}\right] \leq ||f||_{V^{1/2}}\left[1+\frac{K^{1/2}-\lambda^{1/2}}{1-\lambda^{1/2}}\right].$$

Appendix B

Alternative Drift Functions

B.1 Quartic Drift Function

In this section we establish the drift condition for the drift function $V(\mu) = (\mu - \overline{y})^4 + 1$ in order to introduce a drift condition for more general drift functions.

Theorem 7. *The drift condition holds for the* μ*-chain for*

$$V(\mu) = (\mu - \overline{y})^4 + 1, \quad K = L + \rho\omega^4, \quad \lambda = \frac{K}{\omega^4 + 1},$$

provided $\omega > \sqrt[4]{K-1}$ *and* $\alpha > 2$*, where* $\eta := 2d + (n-1)s^2$*,* $\rho := \left(\frac{b^2(\overline{y}-a)^4 + 2b(\overline{y}-a)^2 + 18}{12n^2(\alpha-1)(\alpha-2)}\right)$*,* $L := \rho\eta^2 + 1$*, and* $J := \{\mu \in \mathbb{R} : V(\mu) \leq \omega^4\}$.

Proof.

$$\begin{aligned}
PV(\mu_{m+1}) &= \mathbb{E}\left(\mathbb{E}\left((\mu_{m+1} - \overline{y})^4 + 1 | \tau_m = \tau\right) | \mu_m\right) \\
&= \mathbb{E}\left(\mathbb{E}\left((\mu_{m+1} - \overline{y})^4 | \tau\right) | \mu_m\right) + 1 \\
&= \mathbb{E}\left[(\hat{\mu} - \overline{y})^4 + 6(\hat{\mu} - \overline{y})^2\left(\frac{1}{n\tau + b}\right) + 3\left(\frac{1}{n\tau + b}\right)^2 | \mu_m\right] + 1 \\
&= \mathbb{E}\left[(\hat{\mu} - \overline{y})^4 | \mu_m\right] + 6\mathbb{E}\left[(\hat{\mu} - \overline{y})^2\left(\frac{1}{n\tau + b}\right) | \mu_m\right] + 3\mathbb{E}\left[\left(\frac{1}{n\tau + b}\right)^2 | \mu_m\right] + 1.
\end{aligned}$$

We shall bound each of the above expectations individually, from left to right. The first reduces to:

$$\begin{aligned}
\mathbb{E}\left[(\hat{\mu} - \overline{y})^4 | \mu_m\right] &= \mathbb{E}\left[\left(\frac{b}{n\tau + b}\right)^4 (\overline{y} - a)^4 | \mu_m\right] \\
&= b^4(\overline{y} - a)^4 \mathbb{E}\left[\left(\frac{1}{n\tau + b}\right)^4 | \mu_m\right] \\
&= b^4(\overline{y} - a)^4 \mathbb{E}\left[\left((n\tau)^4 + 4(n\tau)^3 b + 6(n\tau)^2 b^2 + 4n\tau b^3 + b^4\right)^{-1} | \mu_m\right] \\
&\leq b^4(\overline{y} - a)^4 \mathbb{E}\left[\left(\frac{1}{6(n\tau)^2 b^2}\right) | \mu_m\right] \\
&= \frac{b^2(\overline{y} - a)^4}{6n^2} \mathbb{E}\left[\frac{1}{\tau^2} | \mu_m\right].
\end{aligned}$$

We will return to this term after similarly bounding the others. The second bound goes as follows:

$$\begin{aligned}6\mathbb{E}\left[(\hat{\mu}-\overline{y})^2\left(\frac{1}{n\tau+b}\right)|\mu_m\right] &= 6(\overline{y}-a)^2\mathbb{E}\left[\frac{b^2}{(n\tau+b)^3}|\mu_m\right]\\ &= 6b^2(\overline{y}-a)^2\mathbb{E}\left[\left((n\tau)^3+3(n\tau)^2b+3n\tau b^2+b^3\right)^{-1}|\mu_m)\right]\\ &\leq 6b^2(\overline{y}-a)^2\mathbb{E}\left[\frac{1}{3b(n\tau)^2}|\mu_m\right]\\ &= 2\frac{b(\overline{y}-a)^2}{n^2}\mathbb{E}\left[\frac{1}{\tau^2}|\mu_m\right].\end{aligned}$$

We will likewise return to this term after finishing the third and final term, which is more straightforward:

$$3\mathbb{E}\left[\left(\frac{1}{n\tau+b}\right)^2|\mu_m\right] \leq \frac{3}{n^2}\mathbb{E}\left[\frac{1}{\tau^2}\right].$$

After factoring out the common expectation and combining like terms in its coefficient, we are left with the following bound on PV:

$$PV(\mu_{m+1}) \leq \left(\frac{b^2(\overline{y}-a)^4+12b(\overline{y}-a)^2+18}{6n^2}\right)\mathbb{E}\left[\frac{1}{\tau^2}|\mu_m\right]+1. \tag{B.1}$$

Since $\tau|\mu \sim \text{Gamma}(\alpha,\beta)$, we know that $\frac{1}{\tau}|\mu$ follows an inverse gamma distribution. Thus we need only compute the second moment of this distribution to arrive at:

$$PV(\mu_{m+1}) \leq \left(\frac{b^2(\overline{y}-a)^4+12b(\overline{y}-a)^2+18}{6n^2}\right)\frac{(2d+(n-1)s^2+n(\mu_m-\overline{y})^2)^2}{4(\alpha-1)(\alpha-2)}+1. \tag{B.2}$$

Letting $\eta := 2d+(n-1)s^2$ we get

$$PV(\mu_{m+1}) \leq \left(\frac{b^2(\overline{y}-a)^4+12b(\overline{y}-a)^2+18}{24n^2(\alpha-1)(\alpha-2)}\right)\left(\eta+n(\mu_m-\overline{y})^2\right)^2+1.$$

We can bound this in the same way that we bound $(x+y)^2 \leq 2x^2+2y^2$ and arrive at

$$PV(\mu_{m+1}) \leq \left(\frac{b^2(\overline{y}-a)^4+12b(\overline{y}-a)^2+18}{12(\alpha-1)(\alpha-2)}\right)\left(\frac{\eta}{n}\right)^2+\left(\frac{b^2(\overline{y}-a)^4+12b(\overline{y}-a)^2+18}{12(\alpha-1)(\alpha-2)}\right)(\mu_m-\overline{y})^4+1$$

For convenience we define $\rho := \left(\frac{b^2(\overline{y}-a)^4+12b(\overline{y}-a)^2+18}{12(\alpha-1)(\alpha-2)}\right)$ and rewrite the above

$$\begin{aligned}PV(\mu_{m+1}) &\leq \rho\left(\frac{\eta}{n}\right)^2+1+\rho(\mu_m-\overline{y})^4\\ &= \rho\left(\frac{\eta}{n}\right)^2+1-\rho+\rho\left((\mu_m-\overline{y})^4+1\right).\end{aligned}$$

Finally, defining $L := \rho\left(\frac{\eta}{n}\right)^2+1$, we rewrite as

$$PV(\mu_{m+1}) \leq L-\rho+\rho V(\mu_m).$$

For $\mu_m \in J$, we bound

$$PV(\mu_{m+1}) \leq L-\rho+\rho(\omega^4+1) = L+\rho\omega^4 =: K.$$

For $\mu_m \notin J$, we set

$$\lambda := \frac{K}{\omega^4+1} = \frac{L+\rho\omega^4}{\omega^4+1} \geq \frac{L-\rho}{V(\mu_m)}+\rho,$$

since $V(\mu_m)$ is continuous. Since we require $\lambda < 1$, we have that

$$\omega > \sqrt[4]{K-1}.$$

□

Remark: The motivation behind this theorem is that we are now able to apply theorems from Latuszyński et al. (2013) to $f(\mu)=\mu^2$. We claim that the ideas presented above can be extended to higher powers of μ.

B.2 Drift Functions of Even Degree

We now consider the situation where we use $V(\mu) = (\mu - \overline{y})^{2k} + 1,\ k \in \mathbb{N}$.

Theorem 8. *The drift condition for the μ-chain is satisfied if we use*

$$V(\mu) = (\mu - \overline{y})^{2k} + 1, \quad K = L + \omega^{2k}, \quad \lambda = \frac{K}{\omega^{2k} + 1},$$

provided $\omega > \sqrt[2k]{K-1}$, where $\eta := 2d + (n-1)s^2$, $\rho := \sum_{i=0}^{k} \binom{2k}{2i}(2i-1)!!\frac{b^{k-i}\overline{y}^{2(k-i)}}{2\binom{2k-i}{k}\Pi_{j=1}^{k}(\alpha - j)}$, and $L := 1 + \rho\left(\frac{\eta}{n}\right)^k$, for any $k \in \mathbb{N}$.

Proof. First recall that $\tau|\mu \sim \text{Gamma}(\alpha, \beta)$, for

$$\alpha = c + \frac{n}{2}, \quad \beta = d + \frac{(n-1)s^2 + n(\mu - \overline{y})^2}{2},$$

and that

$$n!! = \begin{cases} n \cdot (n-2) \cdot (n-4) \dots 6 \cdot 4 \cdot 2, & n \text{ is even} \\ n \cdot (n-2) \cdot (n-4) \dots 5 \cdot 3 \cdot 1, & n \text{ is odd.} \end{cases} \tag{B.3}$$

We define $-1!! = 0!! = 1$. We will return to this later in our proof. For ease of notation, we remark that since

$$\mu|\tau \sim \mathcal{N}\left(w\overline{y}, \frac{1}{n\tau + b}\right),$$

we have that $\overline{\mu} := \mu - \overline{y} \sim \mathcal{N}\left((w-1)\overline{y}, \frac{1}{n\tau+b}\right)$, and we can thus rewrite

$$\begin{aligned} PV(\mu_{m+1}) &= \mathbb{E}\left[\mathbb{E}\left(\overline{\mu}_{m+1}^{2k}|\tau_m = \tau\right)|\mu_m\right] + 1 \\ &= \mathbb{E}\left[\mathbb{E}\left(\left(\overline{\mu}_{m+1} - \mathbb{E}(\overline{\mu}_{m+1}|\tau) + \mathbb{E}(\overline{\mu}_{m+1}|\tau)\right)^{2k}\right)|\mu_m\right] + 1. \end{aligned}$$

Calling $x := \overline{\mu}_{m+1} - \mathbb{E}(\overline{\mu}_{m+1}|\tau)$, which importantly has mean 0, we can then expand using the binomial theorem as follows:

$$\begin{aligned} PV(\mu_{m+1}) &= \mathbb{E}\left[\sum_{\substack{i=0,\\ i \text{ even}}}^{2k} \binom{2k}{i}\mathbb{E}(x^i|\tau)\mathbb{E}(\overline{\mu}_{m+1}|\tau)^{2k-i}|\mu_m\right] + 1 \\ &= \mathbb{E}\left[\sum_{i=0}^{k} \binom{2k}{2i}\frac{1}{(n\tau + b)^i}(2i-1)!!\mathbb{E}(\overline{\mu}_{m+1}|\tau)^{2k-2i}|\mu_m\right] + 1. \end{aligned}$$

Evaluating this remaining inner expectation yields:

$$\begin{aligned} PV(\mu_{m+1}) &= \mathbb{E}\left[\sum_{i=0}^{k}\binom{2k}{2i}(2i-1)!!\frac{(b\overline{y})^{2(k-i)}}{(n\tau + b)^{2k-i}}|\mu_m\right] + 1 \\ &= \mathbb{E}\left[\sum_{i=0}^{k}\binom{2k}{2i}(2i-1)!!\frac{(b\overline{y})^{2(k-i)}}{\sum_{j=0}^{2k-i}\binom{2k-i}{j}(n\tau)^j b^{2k-i-j}}|\mu_m\right] + 1 \\ &\leq \mathbb{E}\left[\sum_{i=0}^{k}\binom{2k}{2i}(2i-1)!!\frac{(b\overline{y})^{2(k-i)}}{\binom{2k-i}{k}(n\tau)^k b^{k-i}}|\mu_m\right] + 1 \\ &= \mathbb{E}\left[\frac{1}{\tau^k}\sum_{i=0}^{k}\binom{2k}{2i}(2i-1)!!\frac{b^{k-i}\overline{y}^{2(k-i)}}{\binom{2k-i}{k}n^k}|\mu_m\right] + 1. \end{aligned}$$

Since this sum within the expectation is only a constant, we can pull it out, yielding:

$$PV(\mu_{m+1}) \leq 1 + \sum_{i=0}^{k}\binom{2k}{2i}(2i-1)!!\frac{b^{k-i}\overline{y}^{2(k-i)}}{\binom{2k-i}{k}n^k}\mathbb{E}\left[\frac{1}{\tau^k}|\mu_m\right],$$

and we have only to evaluate the k^{th} moment of $\frac{1}{\tau}|\mu_m \sim \text{IG}(\alpha, \beta)$. What results goes as follows:

$$PV(\mu_{m+1}) \leq 1 + \frac{\beta^k}{\Pi_{j=1}^k(\alpha - j)} \sum_{i=0}^{2k} \binom{2k}{2i} (2i-1)!! \frac{b^{k-i}\overline{y}^{2(k-i)}}{\binom{2k-i}{k} n^k}.$$

Calling $\eta := 2d + (n-1)s^2$ and recalling the value of β above, we rewrite our bound as

$$PV(\mu_{m+1}) \leq 1 + \frac{(\eta + n\overline{\mu}_m^2)^k}{2^k} \sum_{i=0}^{k} \binom{2k}{2i} (2i-1)!! \frac{b^{k-i}\overline{y}^{2(k-i)}}{\binom{2k-i}{k} n^k \Pi_{j=1}^k(\alpha - j)},$$

We then further bound PV by bounding this binomial as we do with $(x+y)^k \leq 2^{k-1}(x^k + y^k)$:

$$\begin{aligned} PV(\mu_{m+1}) &\leq 1 + \frac{\left(2^{k-1}\eta^k + 2^{k-1}n^k\overline{\mu}_m^{2k}\right)}{2^k} \sum_{i=0}^{k} \binom{2k}{2i} (2i-1)!! \frac{b^{k-i}\overline{y}^{2(k-i)}}{\binom{2k-i}{k} n^k \Pi_{j=1}^k(\alpha - j)} \\ &= 1 + \left(\eta^k - n^k + n^k V(\mu_m)\right) \sum_{i=0}^{k} \binom{2k}{2i} (2i-1)!! \frac{b^{k-i}\overline{y}^{2(k-i)}}{2\binom{2k-i}{k} n^k \Pi_{j=1}^k(\alpha - j)}. \end{aligned}$$

and we define the constants

$$\rho := \sum_{i=0}^{k} \binom{2k}{2i} (2i-1)!! \frac{b^{k-i}\overline{y}^{2(k-i)}}{2\binom{2k-i}{k} \Pi_{j=1}^k(\alpha - j)}, \quad L := 1 + \rho \left(\frac{\eta}{n}\right)^k$$

in order to rewrite more simply as

$$PV(\mu_{m+1}) \leq L - \rho + \rho V(\mu_m).$$

For $\mu_m \in J$, we bound PV above by

$$PV(\mu_{m+1}) \leq L - \rho + \rho(\omega^{2k} + 1) = L + \rho\omega^{2k} =: K.$$

For $\mu_m \notin J$, we require

$$PV(\mu_{m+1}) \leq L - \rho + \rho V(\mu_m) \leq \lambda V(\mu_m),$$

and solve the right hand inequality for λ. This yields

$$\lambda := \frac{K}{\omega^{2k} + 1} = \frac{L + \rho\omega^{2k}}{\omega^{2k} + 1} \geq \frac{L - \rho}{V(\mu_m)} + \rho,$$

for all $\mu_m \notin J$. Since $\lambda < 1$, it must be true that

$$\omega > \sqrt[2k]{K - 1}.$$

$\square$

One area of future work is to consider drift functions of the form $V(\mu) = (\mu - \overline{y})^{2k} + \alpha(\mu)$, where α is a continuous function of μ. Then it is easy to see the methods we have developed in this paper can be easily extended to drift functions of this form.

Appendix C

Improving the V-norm bounds

Here we provide a theorem which states the conditions on the drift function for the μ-chain the V-norm condition.

Theorem 9. *Using the drift function $V(\mu) = (\mu - \overline{y})^{2k} + 1$, for $k \in \mathbb{Z}$, the V-norm condition is satisfied for $f(\mu) = \mu^j$ if $j \leq k$.*

Proof. Note that (1.4) requires a bound on $||\overline{f}||$, which requires knowledge on $\mathbb{E}_\pi(f)$. This mean is unknown in practice. Instead we bound $||f||_{V^{1/2}}$ and use the following inequality from Appendix A:

$$||\overline{f}||_{V^{1/2}} \leq ||f||_{V^{1/2}} \left[1 + \frac{K^{1/2} - \lambda^{1/2}}{1 - \lambda^{1/2}}\right].$$

It is enough to show that the norm of f is finite, where $f(\mu) = (\mu - \overline{y})^j, j \in \mathbb{Z}$.

$$||f||_{V^{1/2}} = \sup_{\mu \in \mathbb{R}} \frac{|\mu - \overline{y}|^j}{\sqrt{(\mu - \overline{y})^{2k} + 1}} < \infty \quad \text{iff} \quad j \leq k.$$

□

Note that in the above computations, we simplified our work by defining $f(\mu) = (\mu - \overline{y})^j$. The values of the RMSE one obtains when using this value of $||f||_{V^{1/2}}$ are for the random variable $(\mu - \overline{y})^j$. Shifting by $\overline{y}$ does not change the RMSE, since the RMSE is invariant under shifts.

Table C.1 demonstrates that it is possible to reduce $||f||_{V^{1/2}}$ by choosing different drift functions. This leads us to believe that different drift functions could lead to reduced RMSE. B, Theorem 8, we present a theorem that establishes the drift condition for an arbitrary drift function of the form $V(\mu) = (\mu - \overline{y})^{2k}, k \in \mathbb{N}$.

Table C.1: Controlling $||f||_{V^{1/2}}$

j	1	1	0.5	1
k	1	2	2	4
$\|\|f\|\|_{V^{1/2}}$	1	0.707	0.7549	0.7549

This is useful towards our end goal of reducing the upper bound on the RMSE. We now minimize $||f||_{V^{1/2}}$ with respect to μ. Using $f(\mu) = (\mu - \overline{y})^j$, for $j \in \mathbb{Z}$,

$$||f||_{V^{1/2}} = \frac{(\mu - \overline{y})^j}{\sqrt{(\mu - \overline{y})^{2k} + 1}},$$

we set

$$\frac{d}{d\mu}||f||_{V^{1/2}} = \frac{(\mu - \overline{y})^{j-1}\left[j(\mu - \overline{y})^{2k} + j - k(\mu - \overline{y})^{2k}\right]}{\left[(\mu - \overline{y})^{2k} + 1\right]^{3/2}}$$

equal to zero and disregard the case where $\mu = \overline{y}$. We solve for the μ, which we denote $\mu^\star$, that minimizes $||f||_{V^{1/2}}$:

$$\mu^\star := \left(\frac{j}{k-j}\right)^{\frac{1}{2k}} + \overline{y}.$$

Plugging $\mu^\star$ back into $||f||_{V^{1/2}}$ yields the following function of j and k returning the supremum of the norm of f:

$$\sup||f||_{V^{1/2}} = \left(\frac{j}{k-j}\right)^{\frac{j}{2k}} \left(\frac{k-j}{k}\right)^{\frac{1}{2}}.$$

We now fix j and optimize with respect to k:

$$\frac{d}{dk}||f||_{V^{1/2}} = \frac{j\left(\frac{j}{k-j}\right)^{\frac{j}{2k}} \log\left(\frac{j}{k-j}\right)}{2k^2\sqrt{\frac{k}{k-j}}},$$

the critical points of which occur at $k = j$ and $k = 2j$, since $k, j > 0$. As Table C.1 demonstrates, the case where $k = 2j$ yields $||f||_{V^{1/2}} = 1/\sqrt{2}$, the minimum for all $k, j \in \mathbb{R}^+$.

Part III

A Comparison of Programming Languages for MCMC Applications

Chapter 1

Introduction

With the growing use of Markov chain Monte Carlo (MCMC) methods in a wide range of disciplines, the need for efficient programs for MCMC algorithms with fast computation times is becoming more prevalent. In this paper, we will analyze various programming approaches for Markov chains in different statistical models and compare computation speeds in order to determine the most efficient programming language. We will focus on five languages: R, C++ (using the Rcpp package in R), Just Another Gibbs Sampler (JAGS), Julia, and MATLAB. In Section 1.1, we will discuss and define each programming language and all computational tools utilized in our research. In Section 1.2, we will define the Gibbs sampler for the one-sample Normal model, and we will compare the performances of R, Rcpp, JAGS and Julia. Next, we study the performance of a Gibbs sampler for the Bayesian linear regression in Section 2.1. In Section 3 we consider different Bayesian cases of the linear mixed model: we consider models with improper and proper priors, and normal and t-distributed random effects. The last model we will consider is the Probit regression case, and we will investigate the performance of two MCMC algorithms in each of the programming languages in section 4.1. Lastly, we will discuss any limitations we had encountered with the execution of the programming languages in our research. Based on our results, we will provide insight on which coding-language is optimal for MCMC computation.

1.1 Computational Tools

A simple exercise to understand the similarities and differences of R, Rcpp, MATLAB, and Julia is to build a function that computes the n^{th} term of the Fibonacci Sequence. The Fibonacci sequence is defined by the following expression:

$$F_n = F_{n-1} + F_{n-2}, \text{ for } n \geq 2,$$

where $F_0 = 0$ and $F_1 = 1$.

A solution is to define a function recursively with initial conditions and iteratively compute the n^{th} Fibonacci number. To see the differences among R, Rcpp, MATLAB, and Julia, we refer to listings 1-4. There are better ways of coding the Fibonacci sequence; we code in this manner to show the performing gap among these languages.

```r
fibR <- function(n){
  if(n==0){
    return(0)
  }
  if(n==1){
    return(1)
  }
  return(fibR(n-1) + fibR(n-2))
  }
```

Listing 1.1: R Fibonacci Code

The first programming language we used was R, a free, open-source programming language used for statistical computing created by Ross Ihaka and Robert Gentleman. There are a myriad of benefits using R. With its simple syntax, R allows users to easily define functions and construct objects. R contains an abundance of statistical packages from which users can choose to run various functions, and users may also create packages if need be. R can be downloaded at `http:\www.r-project.org/`.

The next programming language we considered in our research was C++ using the Rcpp package in R, allowing users to integrate R and C++. The reason behind our choice to use the Rcpp package rather than pure C++ is we are able to read the data and gather our samples in R with ease. Additionally, Rcpp is favorable to many users as it generally is able to execute codes relatively quickly. Furthermore, a benefit to using Rcpp is that C++ allows users to easily define and control object types, such as vectors and matrices. To install the Rcpp package in R, type `install.packages("Rcpp")` in the R console. In order to utilize matrix computations at an efficient rate, we consider another package within R: *RcppEigen*. The RcppEigen package allowed the C++ environment to work with more linear algebra operations that may not have been available in Rcpp alone. To install RcppEigen, the user would have to type `install.packages("RcppEigen")` in the R console.

```
library(Rcpp)
sourceCpp(code='
          #include <Rcpp.h>

          // [[Rcpp::export]]
          double fibCpp(const double x) {
          if (x == 0) return(0);
          if (x == 1) return(1);
          return (fibCpp(x - 1)) + fibCpp(x - 2);
          }'
)
```

Listing 1.2: Rcpp Fibonacci Code

The third programming language we used is a relatively new programming language, created in 2012 by Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and others, named Julia. Some consider Julia to be the language of the future, and it is meant to have a computational speed comparable to other high-level programming languages, such as C++. Julia's syntax is straightforward and reminiscent of the syntax used in Python and MATLAB. Julia is also free and open-source, and the program can be downloaded at `julialang.org/`.

```
function fibJulia(n)
 if n==0
   return 0
 end
 if n==1
 return 1
 end
 return fibJulia(n-1) + fibJulia(n-2)
 end
```

Listing 1.3: Julia Fibonacci Code

Another programming language in which we ran several of our statistical models was MATLAB. MATLAB is a program created by Jack Little, Steve Bangert and Cleve Moler for technical computing. MATLAB is designed to compute matrix calculations in an optimal manner, but it is not programmed to run for-loops in an efficient manner. To purchase MATLAB, visit `http://www.mathworks.com/`.

```
function f = fibnum(n)
if n == 0
    f = 0;
elseif n == 1
    f = 1;
else
f = fibnum(n-1) + fibnum(n-2);
end
```

Listing 1.4: MATLAB Fibonacci Code

Lastly, we used the program Just Another Gibbs Sampler (JAGS) to run our statistical models. JAGS is different than the previous four languages as it only requires the user to define a statistical model – the MCMC algorithm is chosen by JAGS, not the user. Thus, we will not compare the performance of JAGS to the other four languages; we will only display the computation times. JAGS was ran in R in order to better read our data, and like the Rcpp package; to install the JAGS package in R, type `install.packages("JAGS")` in the R console.

With these five languages, we will explore various MCMC algorithms in order to compare performance times. However, we will only examine the performance of MATLAB in the linear mixed model with normally distributed effects with proper priors and t-distributed effects along with the Probit regression model. In all of our simulations, the number of MCMC chain length 500,000 with a burn-in length of 500,000; that is, we simulated a total of one million MCMC draws for each chain. Each program was ran on a 4th Generation Intel Core i5@ 2.9GHz processor with 12GB of RAM. We now define the Gibbs sampler in the Bayesian one-sample Normal model and compare the performances of the programming languages used in this example.

1.2 One-Sample Normal Model

1.2.1 The Model and the Gibbs Sampler

Consider the data $Y_1, Y_2, ..., Y_n|\mu, \sigma \sim N(\mu, \sigma^2)$ where both μ and σ are unknown. Now to perform a Bayesian analysis on our data, it is common practice to assume the following proper prior distributions of μ and $\tau = \frac{1}{\sigma^2}$:

$$\mu \sim \mathrm{N}\left(a, \frac{1}{b}\right) \quad \text{and} \quad \tau \sim \mathrm{Gamma}\left(c, d\right)$$

where a, b, c and $d \in \mathbb{R}$. When we try to calculate the joint posterior distribution of μ and τ, we are left with an intractable integral. Thus, we must use the posterior conditional distributions, $\tau|\mu$ and $\mu|\tau$, in order to use MCMC methods to gather approximate samples. One is able to show that the conditional distributions are as follows:

$$\tau|\mu, y \sim \mathrm{Gamma}\left(c + \frac{n}{2}, d + \frac{[(n-1)s^2 + n(\bar{y} - \mu)^2]}{2}\right)$$

and

$$\mu|\tau, y \sim \mathrm{N}\left(\hat{\mu}, \frac{1}{n\tau + b}\right)$$

where

$$\hat{\mu} = \hat{\mu}(\tau) = \left(\frac{n\tau}{n\tau+b}\right)\bar{y} + \left(\frac{b}{n\tau+b}\right)a.$$

In this model, s is the sample standard deviation, n is the sample size, and $\bar{y}$ is the sample mean. Now we will use these conditional distributions to create the Gibbs sampler for the one-sample normal model. Starting with our initial pair, (μ_m, τ_m), we must first generate τ_{m+1}, and then use τ_{m+1} to obtain μ_{m+1}. We do this with the following procedure:

1. Draw $\tau_{m+1} \sim \mathrm{Gamma}\left(c + \frac{n}{2}, d + \frac{[(n-1)s^2 + n(\bar{y} - \mu_m)^2]}{2}\right)$
2. Draw $\mu_{m+1} \sim \mathrm{N}\left(\hat{\mu}_m, \frac{1}{n\tau_{m+1}+b}\right)$, where $\hat{\mu}_m = \left(\frac{n\tau_{m+1}}{n\tau_{m+1}+b}\right)\bar{y} + \left(\frac{b}{n\tau_{m+1}+b}\right)a$.

1.2.2 Coding the Gibbs Sampler

Now that we have defined the Gibbs sampler for the one-sample Normal model, we will implement the Gibbs sampler in R, Rcpp, JAGS and Julia.

Within R, there are functions that generate random variates of common distributions that are already pre-defined making it easier to run any statistical model. With a method of generating gamma and normally distributed variates, coding in R wasn't too difficult. Like C++, a variable needs to be defined to store the MCMC chain output.

```r

GibbsNorm = function(iterations, burnin, nthin, mu_prior_precision, mu_prior_mean,
                     tau_prior_shape, tau_prior_rate, mu_initial, tau_initial, data){
  tau.GS <- rep(NA, iterations)
  mu.GS <- rep(NA, iterations)

  y.bar <- mean(data)
  tau.GS[1] <- tau_initial
  mu.GS[1] <- mu_initial
  s <- sd(data)
  n <- length(data)
  post_shape=tau_prior_shape + 0.5 * n

  for(i in 1:burn.in ){
    temptau <- rgamma(1, shape = post_shape,
                      rate = tau_prior_rate + 0.5 * (n * (y.bar - tempmu)^2 + (n-1) * s^2))
    weight <- n * temptau / (n * temptau + mu_prior_precision)
    tempmu <- rnorm(1, mean = weight * y.bar + (1-weight) * mu_prior_mean,
                    sd = 1 / sqrt(n * temptau + mu_prior_precision))
  }


  for(i in 1:iterations){
    for(j in 1:nthin){
      temptau <- rgamma(1, shape = post_shape,
                          rate = tau_prior_rate + 0.5 * (n * (y.bar - tempmu)^2 + (n-1) * s^2))
      weight <- n * temptau / (n * temptau + mu_prior_precision)
      tempmu <- rnorm(1, mean = weight * y.bar + (1-weight) * mu_prior_mean,
                      sd = 1 / sqrt(n * temptau + mu_prior_precision))
    }
  }

  sigma.GS <- 1 / sqrt(tau.GS)

  return(list(mu = mu.GS, tau=tau.GS, sigma=sigma.GS))
}
```

Listing 1.5: One Sample Normal Source R code

GibbsNorm input description:

- `iterations`: An integer value that provides the net length of MCMC chain for main sample
- `burnin`: An integer value that provides the number of draws for MCMC chain to initialize before main sample
- `nthin`: An integer value that provides the number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `mu_prior_precision`: A numeric value that provides the precision parameter for the prior distribution of μ
- `mu_prior_mean`: A numeric value that provides the mean parameter for the prior distribution of μ
- `tau_prior_shape`: A numeric value that provides the shape parameter for the prior distribution of τ
- `tau_prior_rate`: A numeric value that provides the rate parameter for the prior distribution of τ
- `data`: A numeric vector consisting of the observed values from a normal distribution for Bayesian analysis

Since C++ has no immediate package to generate variates from distributions, we have to source functions that generate common distributions that are pre-defined from R. In Listing 6 at line 22, a *call* is initialized to used the functions “rnorm” and “rgamma”. This allows for the C++ function to perform the same tasks as the function in listing 5, but with the speed performance improved.

```
1
2  src='
3  int n_iterations = Rcpp::as<int>(iterations);
4  int burn = Rcpp::as<int>(burnin);
5  int Nthin = Rcpp::as<int>(nthin);
6
7  Rcpp::NumericVector mu(n_iterations);
8  Rcpp::NumericVector tau(n_iterations);
9  Rcpp::NumericVector sigma(n_iterations);
10
11 double tempmu = Rcpp::as<double>(mu_initial);
12 double temptau = Rcpp::as<double>(tau_initial);
13 double s = Rcpp::as<double>(data_sd);
14 double y = Rcpp::as<double>(data_mean);
15 double n = Rcpp::as<double>(data_size);
16 double a = Rcpp::as<double>(mu_prior_mean);
17 double b = Rcpp::as<double>(mu_prior_precision);
18 double c = Rcpp::as<double>(tau_prior_shape);
19 double d = Rcpp::as<double>(tau_prior_rate);
20
21 RNGScope scp;
22 Rcpp::Function rnorm("rnorm");
23 Rcpp::Function rgamma("rgamma");
24
25
```

```
for (int j = 0; j < burn; j++){
temptau = Rcpp :: as<double >(Rcpp :: rgamma(1, (c + ( n / 2.0)),
1.0 / (d + (((n-1.0) * pow(s, 2) + n * (pow(y-tempmu, 2))) / 2.0))));
tempmu = Rcpp :: as<double >(Rcpp :: rnorm(1,
((n * temptau) / (n * temptau + b)) * y + (b / (n * temptau + b)) * a,
1.0 / sqrt(n * temptau + b)));
}

// N_iterations MCMC Chain
for (int i = 0; i < n_iterations; i++){
for (int j = 0; j < Nthin; j++){
temptau = Rcpp :: as<double >(Rcpp :: rgamma(1, (c + (n / 2.0)),
1.0 / (d + (((n-1.0) * pow(s, 2) + n * (pow(y-tempmu, 2))) / 2.0))));
tempmu = Rcpp :: as<double >(Rcpp :: rnorm(1,
((n * temptau) / (n * temptau + b))* y + (b / (n * temptau + b)) * a,
1.0/sqrt(n * temptau + b)));
}
mu[i] = tempmu;
tau[i] = temptau;
sigma[i]= 1.0/sqrt(tau[i]);
}

return Rcpp :: DataFrame :: create (Rcpp :: Named("Mu") = mu,
Rcpp :: Named("Tau") = tau, Rcpp :: Named("Sigma") = sigma);
'

GibbsNormcpp = cxxfunction(signature(iterations = "int",
                                    burnin="int", nthin = "int",mu_prior_mean ="numeric",
                                    mu_prior_precision = "numeric",
                                    tau_prior_shape ="numeric",
                                    tau_prior_rate = "numeric",
                                    mu_initial = "numeric", tau_initial = "numeric",
                                    data_mean = "numeric", data_sd = "numeric",
                                    data_size = "numeric"),
                          src, plugin = "Rcpp")
```

Listing 1.6: One Sample Normal Source Rcpp code

GibbsNormcpp input description:

- `iterations`: An integer value that provides the net length of MCMC chain for main sample
- `burnin`: An integer value that provides the number of draws for MCMC chain to initialize before main sample
- `nthin`: An integer value that provides the number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `mu_prior_precision`: A numeric value that provides the precision parameter for the prior distribution of μ
- `mu_prior_mean`: A numeric value that provides the mean parameter for the prior distribution of μ
- `tau_prior_shape`: A numeric value that provides the shape parameter for the prior distribution of τ
- `tau_prior_rate`: A numeric value that provides the rate parameter for the prior distribution of τ
- `mu_initial`: A numeric value that provides the initial value for MCMC for μ
- `tau_initial`: A numeric value that provides the initial value for MCMC for τ
- `data_mean`: Sample mean of the observed values from a normal distribution for Bayesian analysis
- `data_sd`: Sample standard deviation of the observed values from a normal distribution for Bayesian analysis
- `data_size`: Sample size of the observed values from a normal distribution for Bayesian analysis

Julia is often considered as the language of the future, the creators want users to be able to code with ease with the benefit of running programs as fast as C or C++. Translating the functions as defined in listing 5 to Julia syntax was not too difficult. Coding in Julia was rather simple; there were many similarities in syntax to R such as creating space for the MCMC sample and generating functions. To generate random variates in Julia, it involved using the `rand` function and a distribution of choice. Thus, making it simple to run the Gibbs sampler.

```julia
function GibbsNorm(iterations, burnin, nthin, mu_prior_mean, mu_prior_precision,
    tau_prior_shape, tau_prior_rate, tau_initial, mu_initial, dataset)
    n = length(dataset)
    ybar = mean(dataset)
    s = std(dataset)
    X = fill(0.0, iterations, 3)
    tempmu = mu_initial
    temptau = tau_initial
  post_shape = tau_prior_shape + (n / 2)
    for i in 1:burnin
    rate = tau_prior_rate + (((n-1) * s^2 + n * (ybar - tempmu)^2) / 2.0)
    temptau= rand(Gamma(post_shape, 1.0 / rate))
    w = (n * temptau) / (n * temptau + mu_prior_precision)
    tempmu= rand(Normal((w * ybar) + ((1.0 - w) * mu_prior_mean), 1.0 / sqrt(n * temptau +
    mu_prior_precision) ) )
    end

    for i in 1:iterations
      for j in 1:nthin
    rate = tau_prior_rate + (((n-1) * s^2 + n * (ybar - tempmu)^2) / 2.0)
    temptau= rand(Gamma(post_shape, 1.0 / rate))
    w = (n * temptau) / (n * temptau + mu_prior_precision)
    tempmu= rand(Normal((w * ybar) + ((1.0 - w) * mu_prior_mean), 1.0 / sqrt(n * temptau +
    mu_prior_precision) ) )
      end
      X[i, 2] = temptau
      X[i, 1] = tempmu
    X[i, 3] = 1 / sqrt(temptau)
    end

  return X
end
```

Listing 1.7: One Sample Normal Source Julia code

GibbsNorm (Julia) input description:

- `iterations`: An integer value that provides the net length of MCMC chain for main sample
- `burnin`: An integer value that provides the number of draws for MCMC chain to initialize before main sample
- `nthin`: An integer value that provides the number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `mu_prior_precision`: A numeric value that provides the precision parameter for the prior distribution of μ
- `mu_prior_mean`: A numeric value that provides the mean parameter for the prior distribution of μ
- `tau_prior_shape`: A numeric value that provides the shape parameter for the prior distribution of τ
- `tau_prior_rate`: A numeric value that provides the rate parameter for the prior distribution of τ
- `tau_initial`: A numeric value that provides the initial value for MCMC for τ
- `mu_initial`: A numeric value that provides the initial value for MCMC for μ
- `data`: A numeric vector consisting of the observed values from a normal distribution for Bayesian analysis

Unlike R or Rcpp, JAGS requires the user to provide the prior information and data set information to run a Gibbs sampler. A new script has to be made of type ".jags" in order to initialize the Markov chain. Within R, `jags.model` allows for the user to provide all the information on their model of interest. The functions `update` and `coda.samples` allow for the user to let the chain run a burn-in length and keep their MCMC sample of interest, respectively.

```
cat("
var
    mu_prior_mean, mu_prior_precision , tau_prior_shape , tau_prior_rate , mu, tau , y[N];
    model{
      for (i in 1:N){
        y[i] ~ dnorm(mu, tau)
      }
      mu ~ dnorm(mu_prior_mean , mu_prior_precision)
      tau ~ dgamma(tau_prior_shape ,tau_prior_rate)
    }
    ", file= "onesamplenorm.jags")

jagsfit <- jags.model(file = "onesamplenorm.jags",
                       data = list('mu_prior_mean' = mu.prior.mean,
                                   'mu_prior_precision' = mu.prior.precision ,
                                   'tau_prior_shape' = tau.prior.shape ,
                                   'tau_prior_rate' = tau.prior.rate ,
                                   'y' = outcomes ,
                                   'N' = length(outcomes)
                       ),
                       n.chains = 1, n.adapt = 0)

update(jagsfit , 500000)

MCMC.out <- coda.samples(jagsfit ,
                          var = c("mu","tau"),
                          n.iter = 500000,
                          thin = 1)
```

Listing 1.8: One Sample Normal Source JAGS code

jagsfit input description:

- `mu_prior_mean`: A numeric value that provides the mean parameter for the prior distribution of μ
- `mu_prior_precision`: A numeric value that provides the precision parameter for the prior distribution of μ
- `tau_prior_shape`: A numeric value that provides the shape parameter for the prior distribution of τ
- `tau_prior_rate`: A numeric value that provides the rate parameter for the prior distribution of τ
- `y`: A numeric vector consisting of the observed values from a normal distribution for Bayesian analysis
- `N`: Sample size of the observed values from a normal distribution for Bayesian analysis

1.2.3 Results

For the one-sample Normal model we used a simulated data set to run our Gibbs sampler, and the overall performances Rcpp, JAGS, and Julia are not too different from one another. For one million draws, (500,000 for the MCMC Length and 500,000 for the burn-in length) computational time is relatively low. Each of the languages performed within ten seconds, with Rcpp performing the quickest and R the slowest. Due to how we defined our Gibbs sampler, R has the slowest performance because of how it handles for-loops compared to Rcpp and Julia. The Gibbs sampler requires no matrix calculations so everything done in the coding aspect is scalar computation. We see that JAGS performed second to that of Rcpp. Since writing the model in JAGS is easier to program than the Gibbs sampler, one may consider JAGS as a preliminary tool to obtain approximate samples for Bayesian analysis.

Language	**Average Time (secs)**	**Relative Time**
R	8.981	27.162
Rcpp	0.331	1
Julia	1.141	3.452

Language	**Average Time (sec)**
JAGS	1.026

Next, we consider the Bayesian linear regression model.

Chapter 2

Linear Models

2.1 Linear Regression with Proper Priors

Consider the following data modeling equation:

$$Y|\beta, \sigma \sim N_n\left(X\beta, I_n\sigma^2\right)$$

where Y is an $n \times 1$ vector of observed data, β is an $p \times 1$ vector of regression coefficients, σ is the standard deviation, X is a known $n \times p$ matrix and ϵ denotes the random error where $\epsilon|\sigma \sim N_n(0, I_n\sigma^2)$.

In order to perform a common Bayesian analysis, we will need to assume the commonly used proper priors for β and $\tau = \frac{1}{\sigma^2}$, respectively:

$$\beta \sim N_r(\beta_0, C_0) \perp \tau \sim \text{Gamma}(a, b)$$

Once again, we are left with an intractable integral when trying to calculate the joint posterior distribution; hence, we must construct a Gibbs sampler to obtain approximate samples from the posterior distribution. One can show that the conditional posterior distributons are:

$$\tau|\beta \sim \text{Gamma}\left(a + \frac{n}{2}, b + \frac{SSE + (\beta - \hat{\beta})^T X^T X(\beta - \hat{\beta})}{2}\right) \quad \text{and} \quad \beta|\tau \sim N_p\left(\Sigma \cdot \left[\tau X^T Y + C_0^{-1}\beta_0\right], \Sigma\right)$$

where $\hat{\beta} = (X^T X)^{-1} X^T Y$, $SSE = ||Y - X\hat{\beta}||^2$, and $\Sigma = \left[\tau X^T X + C_0^{-1}\right]^{-1}$
Thus a Gibbs sampler can be defined by beginning with some initial point $(\beta^{(0)}, \tau^{(0)}) \in \mathbb{R}^p \times \mathbb{R}_+$. Then to proceed from $(\beta^{(m)}, \tau^{(m)}) \in \mathbb{R}^p \times \mathbb{R}_+$ to generate $(\beta^{(m+1)}, \tau^{(m+1)})$ for $m \geq 0$, we follow the two steps:

1. Obtain $\tau^{(m+1)} \sim \text{Gamma}\left(a + \frac{n}{2}, b + \frac{SSE + (\beta^{(m)} - \hat{\beta})^T X^T X(\beta^{(m)} - \hat{\beta})}{2}\right)$
2. Obtain $\beta^{(m+1)} \sim N_p\left(M^{(m+1)}, V^{(m+1)}\right)$,
   where $M^{(m+1)} = V^{(m+1)}\left[\tau^{(m+1)} X^T Y + C_0^{-1}\beta_0\right]$ and $V^{(m+1)} = \left[\tau^{(m+1)} X^T X + C_0^{-1}\right]^{-1}$

2.1.1 Coding the Gibbs Sampler

We will now program this Gibbs sampler in R, Rcpp, Julia, and JAGS. This is a slightly more complicated model than the one-sample Normal case, but coding the Gibbs sampler remains to be not too difficult. The coding implementation requires many matrix computation and operations to obtain the posterior MCMC sample.

When coding in R, we run into the dilemma of which function efficiently computes the inverse of a large matrix. Unlike Rcpp and Julia where the computation of the inverse of a matrix is optimized for the language its being executed in, R has more than one way to compute the inverse. R has `solve` function and the composition function `chol2inv(chol())`; the latter function runs quicker than the `solve` function but is less stable. The `solve` function computes the inverse by using matrix algebra–i.e. row operations–but for large matrices there is a significant toll on the functions speed performance. Due to most statistical problems desiring a large sample, the composition function is the preferred choice for overall performance.

```r

Gibbslm <- function(iterations , burnin , nthin , Response , ModelMatrixX ,
                    prior_mean_beta , prior_cov_beta , tau_prior_shape ,
                    tau_prior_rate , start.beta){
  N <- length(Response)
  r <- ncol(ModelMatrixX)

  prior.cov.beta.inv <- chol2inv(chol(prior_cov_beta))
  beta.prior.term <- prior.cov.beta.inv %*% prior_mean_beta
  e <- Response - ModelMatrixX %*% beta.hat
  SSE <- t(e) %*% e
  tau_prior_shape.pos <- tau_prior_shape + N/2.0
  tXX <- t(ModelMatrixX) %*% ModelMatrixX
  tXy <- t(ModelMatrixX) %*% Response

  beta.hat <- chol2inv(chol(tXX)) %*% tXy
  beta <- matrix(NA, nrow = iterations , ncol = r)
  tau <- rep(NA, length = iterations)

  temp_beta <- start.beta
  V_inv <- matrix(NA, nrow = r, ncol = r)

  for(j in 1 : burnin){
    diff <- temp_beta - beta.hat
    postrate <- tau_prior_rate + (SSE + t(diff) %*% tXX %*% diff) / 2.0
    temp_tau <- rgamma(1, shape = tau_prior_shape.pos,
                       rate = postrate)
    V_inv <- temp_tau * tXX + prior.cov.beta.inv
    V <- chol2inv(chol(V_inv))
    temp_beta <- V %*% (temp_tau * tXy + beta.prior.term) + t(chol(V)) %*% rnorm(r)
  }

  for(i in 1 : iterations ){
    for(j in 1:nthin){
      diff <- temp_beta - beta.hat
      postrate <- tau_prior_rate + (SSE + t(diff) %*% tXX %*% diff) / 2.0
      temp_tau <- rgamma(1, shape = tau_prior_shape.pos,
                         rate = postrate)
      V_inv <- temp_tau * tXX + prior.cov.beta.inv
      V <- chol2inv(chol(V_inv))
      temp_beta <- V %*% (temp_tau * tXy + beta.prior.term) + t(chol(V)) %*% rnorm(r)
    }
    beta[i , ] <- temp_beta
    tau[i] <- temp_tau
  }
  sigma <- 1 / sqrt(tau)

  return( list(beta = beta, tau = tau , sigma = sigma))
}
```

Listing 2.1: Linear Regression R code

Gibbslm input description:

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `prior_mean_beta`: A numeric vector for the mean parameter of the normal distribution of β
- `prior_cov_beta`: A numeric matrix for the covariance matrix parameter of the normal distribution of β
- `tau_prior_shape`: A numeric value that provides the shape parameter for the prior distribution of τ

- `tau_prior_rate`: A numeric value that provides the rate parameter for the prior distribution of τ
- `start.beta`: A numeric vector of initial values for MCMC for β

There are no significant limitations to programming the Gibbs sampler in Rcpp as most of the functions that were used for the one-sample Normal scenario carried over to the linear regression case. However, because we are starting to deal with lots of matrices and vector computation we work with RcppEigen which allows for more linear algebra operations.

```

src<- '
using Eigen :: Map;
using Eigen :: MatrixXd;
using Eigen :: VectorXd;
using Eigen :: Vector2d;
using Rcpp :: as;

typedef Eigen :: Map<Eigen :: MatrixXd> MapMatd;
typedef Eigen :: Map<Eigen :: VectorXd> MapVecd;

int n_iterations = Rcpp :: as<int >(iterations);
int burn = Rcpp :: as<int >(burnin);
int nthin = Rcpp :: as<int >(n_thin);

double a = Rcpp :: as<double >(tau_prior_shape);
double b = Rcpp :: as<double >(tau_prior_rate);

Rcpp :: NumericMatrix Xc(ModelMatrixX);
Rcpp :: NumericMatrix CC(Beta_Prior_CovMat);

Rcpp :: NumericVector Yc(Response);
Rcpp :: NumericVector BetaC(Beta_prior_mean);
Rcpp :: NumericVector betainitc(beta_initial);


const MapMatd X(Rcpp :: as<MapMatd>(Xc));
const MapMatd CNot(Rcpp :: as<MapMatd>(CC));

const MapVecd Y(Rcpp :: as<MapVecd>(Yc));
const MapVecd BetaNot(Rcpp :: as<MapVecd>(BetaC));
const MapVecd betainit(Rcpp :: as<MapVecd>(betainitc));

int NRowX = X.rows(), NColX = X.cols();


const MatrixXd C_inv = CNot.inverse();
const MatrixXd tXX = X.transpose() * X;
const MatrixXd tXXinv = tXX.inverse();

const VectorXd tXY = X.transpose() * Y;
const VectorXd betahat = tXXinv * tXY;
const VectorXd diff = Y - X * betahat;

const double SSE = diff.squaredNorm();
const double TauPosShape = a + (NRowX / 2.0);

MatrixXd V(NColX, NColX);
MatrixXd V_inv(NColX, NColX);
MatrixXd betaMCMC(n_iterations , NColX);
MatrixXd tempbeta(1, NColX);

VectorXd eta(NColX);
VectorXd normals(NColX);
VectorXd diffbeta(NColX);
VectorXd tau(n_iterations);
VectorXd sigma(n_iterations);

double rate = 0.0;
double temptau = 1;

```

```
tempbeta = betainit;

RNGScope scp;
Rcpp::Function rnorm("rnorm");
Rcpp::Function rgamma("rgamma");

for(int j = 0; j < burn; j++){
diffbeta = tempbeta - betahat;
rate = b + 0.5 * (SSE + diffbeta.transpose() * tXX * diffbeta);
temptau = Rcpp :: as<double>(Rcpp :: rgamma(1, TauPosShape, 1.0 / rate));
V_inv = temptau * tXX + C_inv;
V = V_inv.inverse();
normals = Rcpp :: as<MapVecd>(Rcpp :: rnorm(NColX));
eta = temptau * tXY + C_inv * BetaNot;
tempbeta = V * eta + V.llt().matrixL() * normals;
}


for(int i = 0; i < n_iterations; i++){
for(int j = 0; j < nthin; j++){
diffbeta = tempbeta - betahat;
rate = b + 0.5 * (SSE + diffbeta.transpose() * tXX * diffbeta);
temptau = Rcpp :: as<double>(Rcpp :: rgamma(1, TauPosShape, 1.0 / rate));
V_inv = temptau * tXX + C_inv;
V = V_inv.inverse();
normals = Rcpp :: as<MapVecd>(Rcpp :: rnorm(NColX));
eta = temptau * tXY + C_inv * BetaNot;
tempbeta = V * eta + V.llt().matrixL() * normals;
}
betaMCMC.row(i) = tempbeta.transpose();
tau[i] = temptau;
sigma[i] = 1.0 / sqrt(temptau);
}

return Rcpp :: DataFrame :: create (Rcpp :: Named("Beta")= betaMCMC,
Rcpp :: Named("Sigma")=sigma);
'

GibbslmCpp = cxxfunction(signature(iterations = "int", burnin = "int",
                                   n_thin = "int", Response = "numeric",
                                   ModelMatrixX = "numeric", Beta_prior_mean = "numeric",
                                   Beta_Prior_CovMat = "numeric",
                                    tau_prior_shape = "numeric",
                                   tau_prior_rate = "numeric",
                                   beta_initial = "numeric"), src, plugin="RcppEigen")
```
Listing 2.2: Linear Regression Rcpp Code

GibbslmCpp:

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `n_thin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `Beta_prior_mean`: A numeric vector for the mean parameter of the normal distribution of β
- `Beta_Prior_CovMat`: A numeric matrix for the covariance matrix parameter of the normal distribution of β
- `tau_prior_shape`: A numeric value that provides the shape parameter for the prior distribution of τ
- `tau_prior_rate`: A numeric value that provides the rate parameter for the prior distribution of τ
- `beta_initial`: A numeric vector of initial values for MCMC for β

Similar to coding the model in Rcpp, Julia had no difficulty in defining a function for the Gibbs sampler. Many of the functions that were used in the one-sample Normal case carried over to the linear regression scenario where the main difference was working with multiple matrices and vectors rather than scalars.

```
function GibbsLM(iterations , burnin , nthin , Response , ModelMatrixX , beta_prior_mean, beta_
    prior_covarmat , tau_prior_shape , tau_prior_rate , BetaInitial , TauInitial)
  n = convert(Float64 , size(ModelMatrixX  , 1))
  m = size(ModelMatrixX , 2)
  Beta = fill(0.0, iterations , m)
  Tau = fill(0.0, iterations , 1)
  tempbeta = fill(0.0, 1, m)
  tempbeta = BetaInitial'
  temptau = TauInitial
  sigma = fill(0.0, iterations)
  BetaHat = fill(0.0,m)
  Rate = 0.0
  V_inv = fill(0.0, m , m)
  eta = fill(0.0, m)
  CholV = fill(0.0, m, m)
  tXX = ModelMatrixX'* ModelMatrixX
  tXy = ModelMatrixX' * Response
  BetaHat = inv(tXX) * transpose(ModelMatrixX) * Response
  SSE = (norm(Response - (ModelMatrixX * BetaHat)))^2
  post_shape = tau_prior_shape + (n / 2.0)

 for i in 1:burnin
    Rate =  norm((tau_prior_rate + ((SSE + (tempbeta - BetaHat') * tXX * transpose((tempbeta -
     BetaHat'))) / 2 )))
    temptau = rand(Gamma(post_shape, (1.0 / Rate) ) )
    V_inv = (temptau) * tXX + inv(beta_prior_covarmat)
    V = inv(V_inv)
    normals = rand(Normal(0,1), m)
    eta = (temptau * tXy) + (inv(beta_prior_covarmat) * beta_prior_mean)
    CholV= transpose(chol(V))
    tempbeta = transpose((V * eta) + (CholV * normals))
  end

 for i in 1:iterations
   for j in 1:nthin
    Rate =  norm((tau_prior_rate + ((SSE + (tempbeta - BetaHat') * tXX * transpose((tempbeta -
     BetaHat'))) / 2 )))
    temptau = rand(Gamma(tau_prior_shape + (n / 2.0), (1.0 / Rate) ) )
    V_inv = (temptau) * tXX + inv(beta_prior_covarmat)
    V = inv(V_inv)
    normals = rand(Normal(0,1), m)
    eta = (temptau * tXy) + (inv(beta_prior_covarmat) * beta_prior_mean)
    CholV= transpose(chol(V))
    tempbeta = transpose((V * eta) + (CholV * normals))
  end
    Beta[i, :] = tempbeta'
    Tau[i] = temptau
    sigma[i] = (1.0 / sqrt(Tau[i]))
  end

return [Beta Tau sigma]
end
```

Listing 2.3: Linear Regression Julia Code

GibbsLM input description:

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model

- `beta_prior_mean`: A numeric vector for the mean parameter of the normal distribution of β
- `beta_Prior_covmat`: A numeric matrix for the covariance matrix parameter of the normal distribution of β
- `tau_prior_shape`: A numeric value that provides the shape parameter for the prior distribution of τ
- `tau_prior_rate`: A numeric value that provides the rate parameter for the prior distribution of τ
- `BetaInitial`: A numeric vector of initial values for MCMC for β
- `TauInitial`: A numeric value for MCMC for τ

In JAGS there was a slight obstacle of how to define the linear model: whether to define the model component wise or define the model in matrix notation. In listing 2.1.1, we define the model component wise; however, if one decided to write the model using matrix notation they would observe a dramatic difference in computational performance. This reason is unknown to us, however we believe that it is due to the intended specific syntax that the creator wanted when implementing the Gibbs sampler.

```
cat( "
    var
      Response[N], MIN[N], prior.mean[P], prior.precision[P, P],
      tau_prior_shape, tau_prior_rate, beta[P], tau;
        model{
          # Likelihood specification
          for(i in 1:N){
          Response[i] ~ dmnorm(mu[i], tau)
          mu[i] <- beta[1] + beta[2] * MIN[i]
          }
          # Prior specification
          beta[] ~ dmnorm(prior.mean[], prior.precision[,])
          tau ~ dgamma(tau_prior_shape, tau_prior_rate)
          sigma <- sqrt(1 / tau)
        }",
     file="LinearRegressionNBA.jags")

jagsfit <- jags.model(file = "LinearRegressionNBA.jags",
                      data = list('Response' = log(NBA.r$PTS),
                                  'MIN' = NBA.r$MIN,
                                  'N' = length(NBA.r$MIN),
                                  'P' = ncol(ModelMatrixX),
                                  'prior.mean' = as.vector(prior.mean.beta),
                                  'prior.precision' = solve(prior.cov.beta),
                                  'tau_prior_shape' = tau.prior.shape,
                                  'tau_prior_rate' = tau.prior.rate),
                      inits = list('beta'= as.vector(beta.hat),'tau'=1),
                      n.chains=1,
                      n.adapt=0
)
update(jagsfit, 500000)

MCMC.out <- coda.samples(jagsfit,
                         var = c("beta", "sigma"),
                         n.iter = 500000,
                         thin = 1)
```
Listing 2.4: Linear Regression JAGS Code

jagsfit input description:

- `Response`: A numeric vector of observed data for linear model
- `MIN`: A numeric vector of observed data for the variable MIN for linear model of NBA
- `N`: Sample size of the observed values

- `P`: The number of columns for the model matrix of linear model, i.e. (Number of predictors used for linear model) + 1
- `prior.mean`: A numeric vector for the mean parameter of the normal distribution of β
- `prior.precision`: A numeric matrix for the covariance matrix parameter of the normal distribution of β
- `tau_prior_shape`: A numeric value that provides the shape parameter for the prior distribution of τ
- `tau_prior_rate`: A numeric value that provides the rate parameter for the prior distribution of τ
- `beta`: A numeric vector of initial values for MCMC for β
- `tau`: A numeric value for MCMC for τ

2.1.2 Results

For the linear regression model, we used a subset of the NBA 2015 Season data; we excluded players who did not play for more than five games and those who did not score any points, on average, per game, leaving us with a sample size of 468 players. With this subset, we used the following linear model equation:

$$\log(PTS_i)|\beta, \tau \sim N\left(\beta_1 + \beta_2 MIN_i, \frac{1}{\tau}\right)$$

where i indicates the player.

As one can see from the table below, Rcpp outperformed both R and Julia by more than thirty seconds and four seconds, respectively. R performed the slowest out of all of the languages. Unlike the one-sample Normal case where R was strictly performing scalar computation, R is now performing matrix calculations. In order to execute this task, R invokes the programming language C which expedites the computation time, however, with R's poor handling of for-loops, its overall performance time is still slow. Although we are uncertain which algorithm JAGS is employing to generate the samples, we see that there is little variation in computation time between R and JAGS in this example.

NBA 2015 Season Data Computation Times:

Language	Average Time (sec)	Relative Time
R	32.564	25.322
Rcpp	1.286	1
Julia	5.452	4.240

NBA 2015 Season Data JAGS Computation Time:

Language	Average Time (sec)
JAGS	29.176

Now we will expand on the linear regression model and introduce the linear mixed model.

Chapter 3

General Linear Mixed Models

Consider the data model equation the general linear mixed model:

$$Y = X\beta + Zu + \epsilon$$

where Y is an $N \times 1$ random vector of observations, X is an $N \times p$ matrix of covariates with rank $(X) = p$, β is a $p \times 1$ vector of regression coefficients, Z is a known, non-zero $N \times q$ matrix, and $\epsilon \sim N_N(0, \sigma_e^2 I)$. The vector of random effects, $u = (u_1, u_2, \ldots, u_q)^T$, may have one of the following assumptions: $u_i \overset{\text{iid}}{\sim} \text{N}(0, \frac{1}{\lambda_u})$ or $u_i \overset{\text{iid}}{\sim} \text{t}_d(0, \frac{1}{\lambda_u})$. We assume that both u and ϵ are independent random vectors. In this paper, we will focus on three cases of the Bayesian linear mixed model: models with improper and proper priors, and models with normally distributed and t-distributed random effects.

3.1 Linear Mixed Model with Improper Priors

First we will consider a linear mixed model with improper priors. Here we will assume the following priors for $(\beta, \lambda_e, \lambda_u)$:

$$\beta \sim \text{flat prior} \quad \lambda_e \sim \text{Gamma}^*(a_e, b_e) \quad \lambda_u \sim \text{Gamma}^*(a_u, b_u)$$

where

$$\text{Gamma}^*_{a,b} \propto x^{a-1} e^{-bx}$$

and both a and b can be positive or negative.

For the purpose of this paper we will use the three-block Gibbs sampler as defined in Román and Hobert's (2012). Let $\lambda = (\lambda_e \lambda_u)^T$ and $\theta = (\beta^T u^T)^T$. The basic idea is to use $(\lambda^{'}, \theta^{'})$ to generate $(\lambda, \theta^{'})$ followed by using $(\lambda, \theta^{'})$ to generate (λ, θ). To obtain (λ, θ) given $(\lambda^{'}, \theta^{'})$ we proceed with the following:

1. Obtain $\lambda_e | \theta', y \sim \text{Gamma}\left(a_e + \frac{N}{2}, b_e + \frac{\|y - W\theta'\|^2}{2}\right)$ where $W = (X\ Z)$ such that $X\beta + Zu = W\theta$.

2. If $b_u + \frac{\|u'\|^2}{2}$ is positive, generate $\lambda_u | \theta', y \sim \text{Gamma}\left(a_u + \frac{q}{2}, b_u + \frac{\|u'\|^2}{2}\right)$. If $b_u + \frac{\|u'\|^2}{2}$ equates to zero, then generate $\lambda_u | \theta', y \sim \text{Gamma}\,(a, b)$ for $a, b > 0$.

3. Next, we must generate $\theta|\lambda$ from a $p + q$- dimensional Normal Distribution with the following mean vector and covariance matrix, respectively:

$$E(\theta|\lambda) = \begin{bmatrix} (X^T X)^{-1} X^T \big(I - \lambda_e Z \tilde{Q}_\lambda^{-1} Z^T P^\perp\big) y \\ \lambda_e \tilde{Q}_\lambda^{-1} Z^T P^\perp y \end{bmatrix}$$

$$\text{Var}(\theta|\lambda) = \begin{bmatrix} (\lambda_e X^T X)^{-1} + (X^T X)^{-1} X^T Z \tilde{Q}_\lambda^{-1} Z^T X (X^T X)^{-1} & -(X^T X)^{-1} X^T Z \tilde{Q}_\lambda^{-1} \\ -\tilde{Q}_\lambda^{-1} Z^T X (X^T X)^{-1} & \tilde{Q}_\lambda^{-1} \end{bmatrix}$$

where $P^\perp = I - X(X^T X)^{-1} X^T$ and $\tilde{Q}_\lambda = \lambda_e Z^T P^\perp Z + I_q \lambda_u^{-1}$.

3.2 Linear Mixed Model with Proper Priors

3.2.1 Normally Distributed Random Effects

Consider the Linear Mixed Model except with proper priors from Román and Hobert's (2015). We begin by using the following proper priors for $(\beta, \lambda_e, \lambda_u)$:

$$\beta \sim N_p(\beta_0, \Sigma_\beta) \perp \lambda_e \sim \text{Gamma}(a_e, b_e) \perp \lambda_u \sim \text{Gamma}(a_u, b_u)$$

What follows is a posterior distribution that is intractable. Román and Hobert's (2015) provides a Gibbs sampler with blocks λ and θ based off the conditional posterior distributions. We will use the same process as the improper prior case, however, there is no need to perform the checks to obtain $\lambda_u|\theta'$ as that issue only occurs when we have improper prior distributions. Thus we can proceed with generating $\lambda|\theta'$:

1. Draw
$$\lambda_e|\theta' \sim \text{Gamma}\left(a_e + \frac{N}{2}, b_e + \frac{\|y - W\theta'\|^2}{2}\right)$$

2. Draw
$$\lambda_u|\theta' \sim \text{Gamma}\left(a_u + \frac{q}{2}, b_u + \frac{\|u'\|^2}{2}\right)$$

Now we will generate $\theta|\lambda$ from a $p+q$- dimensional Normal Distribution with the following mean vector and covariance matrix, respectively:

$$E_\pi(\theta|\lambda) = \begin{bmatrix} T_\lambda^{-1}(\lambda_e X^T y + \Sigma_\beta^{-1}\beta_0) - \lambda_e^2 T_\lambda^{-1} X^T Z Q_T^{-1} Z^T R_\lambda \\ \lambda_e Q_T^{-1} Z^T R_\lambda \end{bmatrix}$$

$$\text{Var}_\pi(\theta|\lambda) = \begin{bmatrix} T_\lambda^{-1} + \lambda_e^2 T_\lambda^{-1} X^T Z Q_T^{-1} Z^T X T_\lambda^{-1} & -\lambda_e T_\lambda^{-1} X^T Z Q_T^{-1} \\ -\lambda_e Q_T^{-1} Z^T X T_\lambda^{-1} & Q_T^{-1} \end{bmatrix}$$

where $T_\lambda = \lambda_e X^T X + \Sigma_\beta^{-1}$, $R_\lambda = M_\lambda y - X T_\lambda^{-1} \Sigma_\beta^{-1} \beta_0$, $M_\lambda = I - \lambda_e X T_\lambda^{-1} X^T$, and $Q_T = \lambda_e Z^T M_\lambda Z + \lambda_u I_q$.

3.2.2 t-Distributed Random Effects

Suppose instead of Normally distributed random effects, we assume that the random effects follow a t-distribution. The model we would be consider is:

$$Y|\beta, u, \lambda_e, \lambda_u \sim N_n(X\beta + Zu, \lambda_e^{-1} I)$$

$$\beta \sim N_p(\mu_\beta, \Sigma_\beta), \lambda_e \sim \text{Gamma}(a_e, b_e), u_i|\lambda_u \overset{\text{iid}}{\sim} t_d(0, \lambda_u^{-1})$$

$$\lambda_u \sim \text{Gamma}(a_u, b_u)$$

where Y is an $N \times 1$ data vector, X is a known $N \times p$ matrix, Z is a known $N \times q$ matrix, β is a $p \times 1$ vector of regression coefficients, $u = (u_1, u_2, \ldots, u_q)^T$ is the vector of random effects, λ_e is a precision parameter, and λ_u^{-1} is the squared scale parameter.

To perform Bayesian analysis, we use the Gibbs sampler from Román et al.'s (2016). The authors introduce a new variable, η, to the previous model in order to run the Gibbs sampler. What follows is an indirect route that requires obtaining draws from the posterior distribution of the slightly more complicated model:

$$Y|\beta, u, \lambda_e, \lambda_u \sim N_n(X\beta + Zu, \lambda_e^{-1} I)$$

$$\beta \sim N_p(\mu_\beta, \Sigma_\beta), \lambda_e \sim \text{Gamma}(a_e, b_e), u_i|\lambda_u \overset{\text{iid}}{\sim} t_d(0, \lambda_u^{-1})$$

$$\lambda_u \sim \text{Gamma}(a_u, b_u), \eta_i \sim \text{Gamma}\left(\frac{d}{2}, \frac{d}{2}\right) \quad \text{for } i = 1, 2, \ldots, q$$

where d denotes the degrees of freedom.

Their Gibbs sampler involves three blocks: $\eta = (\eta_1, \eta_2, \ldots, \eta_q)$ $\lambda = (\lambda_e, \lambda_u)$ and $\theta = (\beta^T u^T)^T$. Given $(\lambda^m, \eta^m, \theta^m)$, the steps to obtain $(\lambda^{m+1}, \eta^{m+1}, \theta^{m+1})$ is as follows:

1. Draw $\lambda_e | \eta, \theta \sim \text{Gamma}\left(a_e + \frac{N}{2}, b_e + \frac{||y - W\theta||^2}{2}\right)$

   where $||y - W\theta||$ is the Frobenius norm.

2. Draw $\lambda_u | \eta, \theta \sim \text{Gamma}\left(a_u + \frac{q}{2}, b_u + \frac{||D_\eta^{1/2} u||^2}{2}\right)$

   where $D_\eta = \text{diag}\,(\eta_1, \eta_2, \ldots, \eta_q)$.

3. Draw $\eta_i | \theta, \lambda$ independently from Gamma $\left(\frac{d+1}{2}, \frac{d + \lambda_u u_i^2}{2}\right)$ for $i = 1, \ldots, q$

4. Generate $\theta | \lambda, \eta$ from a multivariate Normal distribution with the following mean vector and covariance matrix, respectively:

$$E(\theta|\lambda, \eta) = \begin{bmatrix} T_\lambda^{-1}(\lambda_e X^T y + \Sigma_\beta^{-1}\mu_\beta) - \lambda_e^2 T_\lambda^{-1} X^T Z Q_{\lambda,\eta}^{-1} Z^T (M_\lambda y - X T_\lambda^{-1} \Sigma_\beta^{-1} \mu_\beta) \\ \lambda_e Q_T^{-1} Z^T (M_\lambda y - X T_\lambda^{-1} \Sigma_\beta^{-1} \mu_\beta) \end{bmatrix}$$

$$\text{Var}(\theta|\lambda, \eta, y) = \begin{bmatrix} T_\lambda^{-1} + \lambda_e^2 T_\lambda^{-1} X^T Z Q_{\lambda,\eta}^{-1} Z^T X T_\lambda^{-1} & -\lambda_e T_\lambda^{-1} X^T Z Q_{\lambda,\eta}^{-1} \\ -\lambda_e Q_{\lambda,\eta}^{-1} Z^T X T_\lambda^{-1} & Q_{\lambda,\eta}^{-1} \end{bmatrix}$$

3.3 Results

For the analysis of the linear mixed model, we will consider the same subset of the NBA 2015 season data used in the linear regression example. The random intercept model that we consider is:

$$\log(PTS_{ij}) = \beta_1 + \beta_2 MIN_{ij} + u_j + \epsilon_i$$

where i indicates the player, j indicates the player's team, and the random effect u_j is team affiliation.

We consider analyzing the performance of the computing languages in the following scenarios: improper priors, normally distributed random effects, and t-distributed random effects

3.3.1 LMM with Normal Random Effects and Improper Priors

For the improper prior case, we see that Rcpp continues to outperform both Julia and R. Rcpp can compute the Markov Chain in this model about twenty seconds faster than Julia. Relatively, each programming language does not vary too much from one another as R only takes four times as long to finish one iteration. The computation times for the linear mixed model are comparatively slower than the results seen in the one-sample Normal case due to the heavy matrix calculations involved. Lastly, JAGS took the longest to finish the calculations, taking roughly eight minutes to generate the approximate samples.

Linear Mixed Model with improper prior computation times:

Language	Average Time (sec)	Relative Time
R	165.84	4.660
Rcpp	35.585	1
Julia	52.675	1.480

Language	Average Time (sec)
JAGS	486.567

3.3.2 LMM with Normal Random Effects and Proper Priors

In this example, all languages took roughly the same amount of time to generate our samples as in the improper case. Rcpp continued to have the fastest computation time with Julia following closely behind. In the proper case, our programs, with the exception R, took at most twenty more seconds to compute than in the improper case. For an additional language comparison, we programmed the Gibbs sampler in MATLAB, which took longer to compute than R, Rcpp and Julia. Although MATLAB is programmed to complete matrix calculations quickly, MATLAB is not equipped to handle many nested for-loops in an optimal manner. Since our program consisted of nested for-loops, MATLAB had a subpar performance. Furthermore, R used C to perform the matrix calculations, thus it relatively twice as long as Rcpp and Julia to compute the approximate samples. Lastly, JAGS had the slowest computation time, taking around eight minutes. All of the languages had a similar performance in the improper and proper case.

Language	**Average Time (sec)**	**Relative Time**
R	167.856	2.925
Rcpp	57.391	1
MATLAB	236.346	4.118
Julia	71.335	1.243

Language	**Average Time (sec)**
JAGS	499.2

3.3.3 LMM with t-Distributed Random Effects and Proper Priors

Unlike the previous linear mixed model examples, all programming languages, with the exception of JAGS, took more than twenty minutes to complete one iteration. Julia had a faster computation time than Rcpp in this model. One reason being is Rcpp had to call R to be able to run numerous functions, such as the Frobenius norm function. We also included an extra for-loop in the code in order to generate D_η. These factors combined hindered Rcpp from outperforming the other languages. Moreover, R took almost four times as long to compute our samples than Julia. As in the aforementioned linear mixed model cases, the algorithms consisted of a lot of matrix calculations, which slowed computation time for all of the languages. Here we observe that performance of JAGS was also slower in this model, taking an additional three minutes to produce our samples.

Language	**Average Time (sec)**	**Relative Time**
R	5555.864	3.906
Rcpp	1654.690	1.163
MATLAB	1760.597	1.238
Julia	1422.387	1

Language	**Average Time (sec)**
JAGS	663.481

Now we consider our last model, the Probit regression model.

Chapter 4

Probit Regression

4.1 The Model and the MCMC Algorithms

Consider $Y_1, \ldots, Y_n$ are independent Bernoulli random variables such that $P(Y_i = 1) = F(x_i^T \beta)$ where x_i is a $p \times 1$ vector of known covariates associated with Y_i, β is a $p \times 1$ vector of unknown regression coefficients and $F(\cdot)$ denotes the cumulative distribution function (CDF). It follows that

$$P\left(Y_1 = y_1, \ldots, Y_n = y_n | \beta\right) = \prod_{i=1}^{n} \left[F(x_i^T \beta)\right]^{y_i} \left[1 - F(x_i^T \beta)\right]^{1-y_i}.$$

One of the common methods for modeling Binary data is to consider a Probit regression model. For a Probit Regression Model, we let $F(\cdot)$ be the standard normal CDF; i.e. $F(x) = \Phi(x)$. A useful method for making inferences on β by Bayesian analysis is to consider a flat prior on β.

Albert and Chib's (1993) algorithm (henceforth, the "AC algorithm") provide a way of obtaining a MCMC sample from the posterior distribution of β. To transition from current state β to the new state β', one must consider the following steps:

1. Draw $z_1, \ldots, z_n$ independently with $z_i \sim TN(x_i^T \beta, 1, y_i)$
2. Draw $\beta' \sim N_p\left((X^T X)^{-1} X^T z, (X^T X)^{-1}\right)$

A modified version of the AC algorithm is Liu and Wu's (1999) PX-DA algorithm which also provides a method of obtaining a MCMC sample from the posterior distribution of β. It involves one more additional step sandwiched between the steps in the AC algorithm. To transition from current state β to the new state β', one must consider the following steps:

1. Draw $z_1, \ldots, z_n$ independently with $z_i \sim TN(x_i^T \beta, 1, y_i)$
2. Draw $g^2 \sim \text{Gamma}\left(\frac{n}{2}, \frac{1}{2}\sum_{i=1}^{n}(z_i - x_i^T (X^T X)^{-1} X^T z)^2\right)$
3. Set $z' = (gz_1, \ldots, gz_n)^T$
4. Draw $\beta' \sim N_p\left((X^T X)^{-1} X^T z', (X^T X)^{-1}\right)$

4.2 Coding the AC Algorithm

We will show the code for the AC algorithm and not the PX-DA algorithm in this section. For the code for the PX-DA algorithm refer to the appendix.

Coding the AC algorithm in R, Rcpp, Julia, and MATLAB proved to be a challenge. Because the method involved generating random truncated normal variates, there were many barriers to defining the function. In addition, the model requires several nested for-loops causing many of the functions to slow significantly compared to the previous models.

```
GibbsProbit = function(iterations, burnin, nthin, Response, ModelMatrixX, betainitial){
  X <- ModelMatrixX
  Y <- Response
  tXX <- t(X) %*% X
```

```r
  txxinv <- solve(tXX)
  n <- length(Y)
  p <- ncol(X)

  BetaMCMC <- matrix(NA, nrow = iterations, ncol = p)
  tempbeta <- betainitial
  z <- matrix(NA, n,1)
  V <- t(chol(txxinv))
  for(k in 1:burnin){
    for(j in 1:n){
      center <- t(X[j,]) %*% tempbeta
      if(Y[j] == 0){
        z[j] <- rtruncnorm(1, a = -Inf, b = 0, mean = center, sd = 1)
      }
      if(Y[j] == 1){
        z[j] <- rtruncnorm(1, a = 0, b = Inf, mean = center, sd = 1)
      }
    }
    betahat <- txxinv %*% t(X) %*% z
    tempbeta <- betahat + V %*% rnorm(p)
  }

  for(i in 1:(iterations)){
    for(k in 1:nthin){
      for(j in 1:n){
        center <- t(X[j,]) %*% tempbeta
        if(Y[j] == 0){
          z[j] <- rtruncnorm(1, a = -Inf, b = 0, mean = center, sd = 1)
        }
        if(Y[j] == 1){
          z[j] <- rtruncnorm(1, a = 0, b = Inf, mean = center, sd = 1)
        }
      }
      betahat <- txxinv %*% t(X) %*% z
      tempbeta <- betahat + V %*% rnorm(p)
    }
    BetaMCMC[i,] <- tempbeta
  }
  return(as.mcmc(BetaMCMC))
}
```

Listing 4.1: Probit Regression R code

GibbsProbit input description:

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `betainitial`: A numeric vector of initial values for MCMC of β

For R, there were several ways of generating truncated normals. There are two packages that we've worked with `msm` and `truncnorm` to generate truncated normals. In `truncnorm`, we use the function `rtruncnorm` to generate the truncated normals that are neccessary for the algorithm.

```cpp
src<- '
using Eigen::Map;
using Eigen::MatrixXd;
using Eigen::VectorXd;
using Eigen::Vector2d;
using Rcpp::as;

typedef Eigen::Map<Eigen::MatrixXd> MapMatd;
typedef Eigen::Map<Eigen::VectorXd> MapVecd;

int MCMCiter = Rcpp::as<int>(iterations);
int burnin = Rcpp::as<int>(Burnin);
int n_thin = Rcpp::as<int>(nthin);

Rcpp::NumericMatrix Xc(ModelMatrixX);
Rcpp::NumericVector Yc(Response);
Rcpp::NumericVector BetaInitialc(BetaInitial);

const MapMatd X(Rcpp::as<MapMatd>(Xc));
const MapVecd Y(Rcpp::as<MapVecd>(Yc));
const MapVecd Betainitial(Rcpp::as<MapVecd>(BetaInitialc));

int n = X.rows();
int p = X.cols();

const MatrixXd tXX = X.transpose() * X;
const MatrixXd tXXinverse = tXX.inverse();

MatrixXd betaMCMC(MCMCiter, p);
MatrixXd V(p, p);

VectorXd tempbeta = Betainitial;
VectorXd Z(n);
VectorXd betahat(p);
VectorXd normals(p);

double center = 0.0;

V = tXXinverse.llt().matrixL();

RNGScope scp;

Rcpp::Function rtnorm("rtnorm");
Rcpp::Function rnorm("rnorm");
Rcpp::Function dnorm("dnorm");
Rcpp::Function pnorm("pnorm");

for(int k = 0; k < burnin; k++){
for(int j = 0; j < n; j++){
center = X.row(j) * tempbeta;

if(Y[j] == 0.0){
Z[j] = as<double>(rtnorm(1, center, 1, R_NegInf, 0));
}

if(Y[j] == 1.0){
Z[j] = as<double>(rtnorm(1, center, 1, 0, R_PosInf));
}
}

betahat = tXXinverse * X.transpose() * Z;
normals = Rcpp::as<MapVecd>(Rcpp::rnorm(p));
tempbeta = betahat + V * normals;
}

for(int i = 0; i < MCMCiter; i++){
for(int k = 0; k < n_thin; k++){
for(int j = 0; j < n; j++){
center = X.row(j) * tempbeta;

if(Y[j] == 0.0){
Z[j] = as<double>(rtnorm(1, center, 1, R_NegInf, 0));
}

if(Y[j] == 1.0){
Z[j] = as<double>(rtnorm(1,center,1,0, R_PosInf));
}
}

betahat = tXXinverse * X.transpose() * Z;
normals = Rcpp::as<MapVecd>(Rcpp::rnorm(p));
tempbeta = betahat + V * normals;
}
betaMCMC.row(i) = tempbeta.transpose();
}

return Rcpp::DataFrame::create(Rcpp::Named("Beta") = betaMCMC);
'
GibbsProbitcpp = cxxfunction(signature(iterations = "int", Burnin = "int", nthin = "int",
                                        Response = "numeric", ModelMatrixX = "numeric",
                                        BetaInitial = "numeric"), src, plugin="RcppEigen")
```

Listing 4.2: Probit Regression Rcpp code

GibbsProbitcpp input description:

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `Betainitial`: A numeric vector of initial values for MCMC of β

In Rcpp, there were several complications; mainly sourcing the functions in R into Rcpp. For example, if one were to use the `rtruncnorm` function when defining the function in Rcpp, a load of errors will present itself in the compilation stage stating the the package is not available in Rcpp. This led for us to use the function `rtnorm` from the `msm` package, which integrated well with Rcpp. However, when we tested the function, we noticed a significant toll on computation time. It took nearly 1 second to generate a truncated normal in Rcpp compared to generating it in R which took about 0.001 of a second. This proved to be a big handicap when working with Rcpp for the probit model.

```
cat( "
     var
      Response[N], ModelMatrixX[N,P], beta[P], lowerbd, upperbd;
      model{
        for(i in 1:N){
          Y[i] ~ dbern(q[i])
          q[i] <- phi(ModelMatrixX[i,]%*% beta[])
        }

        for(i in 1:P){
          beta[i] ~ dnorm(0,1/var) #pseudo SIR prior
        }
     }", file="ProbitRegressionImproper.jags")
jagsfit <- jags.model(file = "ProbitRegressionImproper.jags",
                      data = list('Response' = Y,
                                  'ModelMatrixX' = X,
                                  'N' = length(Y),
                                  'P' = ncol(X),
                                  'var' = 100000000),
                      inits = list('beta'= rep(0, ncol(X))),
                      n.chains=1,
                      n.adapt=0
)

update(jagsfit, 100)
MCMC.out <- coda.samples(jagsfit,
                         var = c("beta"),
                         n.iter = 1000000,
                         thin = 1)
```

Listing 4.3: Probit Regression JAGS code

jagsfit input description:

- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `N`: Sample size of the observed values
- `P`: The number of columns for the model matrix of linear model, i.e. (Number of predictors used for linear model) + 1
- `var`: A numeric value that provides the prior variance for the distribution of β
- `beta`: A numeric vector of initial values for MCMC of β

Coding the Probit model in JAGS was relatively easy to write up. However, because there is no *flat prior* distribution in the directory, we have to consider a pseudo-flat prior for the analysis. This means that were running a different Markov Chain than the one stated in the AC algorithm. So comparisons of JAGS with the other languages is not a true fair comparison, despite the coding ease.

```
function DAProbitModel(iterations, burn_in, nthin, Response, ModelMatrixX, startbeta)
  n = size(ModelMatrixX, 1)
  p = size(ModelMatrixX, 2)
  BetaMCMC = fill(0.0, iterations, p)
  TempBetaMCMC = startbeta
  z = fill(0.0, n, 1)
  txx = transpose(ModelMatrixX) * ModelMatrixX
  txxinverse = inv(txx)
  V = transpose(chol(txxinverse))

  for i in 1:burn_in
      for j in 1:n
        center = ModelMatrixX[j, :] * TempBetaMCMC
        if (Response[j] == 0)
            z[j] = rand(Truncated(Normal(center[1], 1), -Inf, 0.0))
          end
        if (Response[j] == 1)
            z[j] = rand(Truncated(Normal(center[1], 1), 0.0, Inf))
          end
       end
       BetaHat = txxinverse * transpose(ModelMatrixX) * z
     TempBetaMCMC = BetaHat + (V * rand(Normal(),p))
  end


  for i in 1:iterations
    for k in 1:nthin
      for j in 1:n
        center = ModelMatrixX[j, :] * TempBetaMCMC
        if (Response[j] == 0)
            z[j] = rand(Truncated(Normal(center[1], 1), -Inf, 0.0))
          end
        if (Response[j] == 1)
            z[j] = rand(Truncated(Normal(center[1], 1), 0.0, Inf))
          end
       end

       BetaHat = txxinverse * transpose(ModelMatrixX) * z
       TempBetaMCMC = BetaHat + (V * rand(Normal(),p))
    end
  BetaMCMC[i,:] = transpose(TempBetaMCMC)
  end

  return BetaMCMC
end
```

Listing 4.4: Probit Regression Julia code

DAProbitModel input description

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `startbeta`: A numeric vector of initial values for MCMC of β

In Julia, the obstacle of generating truncated normals was non-existent. Using pre-defined functions that allow us to generate truncated random variates, we were able to write the function with ease. Furthermore, there was no significant toll in computing performance in order to generate the truncated normals.

```
function [BetaMCMC] = ProbitDA(iterations, burnin, nthin, Response, ModelMatrixX, startbeta)
    n = length(Response);
    X = ModelMatrixX;
    y = Response;
    p = size(X,2);
    BetaMCMC = repmat(0.0, iterations, p);

    tempbeta = startbeta;

    z = repmat(0.0, n,1);
    znew = repmat(0.0, n, 1);
    tXX = X' * X;
    txxinv = inv(tXX);
    V = transpose(chol(txxinv));

    for i = 1:burnin
        for j = 1:n
        center = X(j,:) * tempbeta;
         pd = makedist('Normal', center, 1);
            if(y(j) == 0)
                z(j) = random(truncate(pd,-inf,0));
            end
            if(y(j) == 1)
                z(j) = random(truncate(pd,0, inf));
            end
        end

        betahat = txxinv * X' * z;
        tempbeta = betahat + (V * normrnd(0,1,p,1));
    end


    for i = 1:iterations
        for nth= 1:nthin
            for j = 1:n
                center = X(j,:) * tempbeta;
                pd = makedist('Normal', center, 1);
                if(y(j) == 0)
                    z(j) = random(truncate(pd,-inf,0));
                end
                if(y(j) == 1)
                    z(j) = random(truncate(pd,0, inf));
                end
            end


            betahat = txxinv * X' * z;
            tempbeta = betahat + (V * normrnd(0,1,p,1));
        end
        BetaMCMC(i,:) = tempbeta;
    end
```

```
end
```
Listing 4.5: Probit Regression MATLAB code

Due to the extreme similarities between Julia and MATLAB, translating the code was not too difficult. However, generating truncated observations had to be implemented differently. It involved defining the distribution every single time we have to draw from a different truncated normal based on the algorithm. This placed a significant toll on the speed performance for MATLAB, slowing the function to draw a single MCMC observation at a rate of approximately 1 per second. Which is too slow to even consider running since we want samples of size 500,000 with a 500,000 burn-in sample.

4.3 AC and PX-DA Algorithm Results

The data set used for the AC and PX-DA algorithm was the Pima Indians Diabetes data set from the National Institute of Diabetes and Digestive and Kidney Diseases. The data set is comprised of 768 Pima Indian females, at least 21 years old, and it records the presence of diabetes in each participant. If a participant tested positive for diabetes, she was given a class value of 1, otherwise she was given a class value of 0. Below is the Probit Regression model equation we used for this example:

$$P(Y_i = 1|\beta) = \Phi(\beta_1 + \beta_2 glucose_i) \ .$$

For the AC Algorithm, Julia outperformed all of the other languages. One reason for this result is Julia has a predefined truncated normal distribution function, whereas neither MATLAB nor Rcpp contains a function for this distribution. This limitation forced us to find other methods to create a truncated normal distribution. In MATLAB, we were forced to define a truncated normal distribution within our for-loops. With a MCMC length of 500,000, a burn-in length of 500,000 and a sample size of 768, we had to define the truncated normal function 768,000,000 times in one iteration. This subsequently slowed our program profoundly, forcing us to not consider the computation time. Similarly, to use a truncated normal distribution function in Rcpp, we had to continuously call R. Once again, this slowed Rcpp's overall performance; one iteration took over sixteen hours to finish. Although R performed as expected, it is important to note that R took over two hours to execute the same task that Julia completed in five minutes. For the AC Algorithm, Julia was the optimal choice of programming language, even outperforming JAGS which is most likely using a different Markov Chain.

AC Algorithm Results:

Language	**Average Time (sec)**	**Relative Time**
R	9528.818	29.092
Rcpp	59018.580	180.185
MATLAB	Too Slow	-
Julia	327.545	1

AC Algorithm JAGS Results:

Language	**Average Time (sec)**
JAGS	664.02

For the PX-DA Algorithm, we saw similar results to that of the AC Algorithm with Julia having the best performance compared to R and Rcpp. While using this algorithm, we still encounter the issue of constantly calling R in order to generate our truncated normals in Rcpp. Thus, Rcpp still performs very slowly. Since the AC and PX-DA Algorithms are two algorithms for the same model, there is no need to run JAGS for the PX-DA algorithm. While the PX-DA algorithm is a more efficient algorithm to generate approximate samples for the Probit Regression model, it has a longer computation time than the AC algorithm. However, the PX-DA algorithm allows the MCMC chain to converge more rapidly. Hence we are left with a trade-off: a faster computation time with a longer MCMC chain or a slower computation time with faster convergence, ultimately requiring a smaller MCMC chain.

PX-DA Algorithm Results:

Language	**Average Time (sec)**	**Relative Time**
R	25080.408	4.005
Rcpp	61930.8	9.890
Julia	6261.812	1

4.4 Limitations/Conclusion

Throughout our research, we discovered each programming language had its benefits and limitations. The greatest benefit of R is its simple syntax and many functions, making it very easy to code the Gibbs samplers in each statistical model. Also, R's ability to call C for matrix calculations increases R's performance, making R a favorable choice of programming language. However, when the dimensions of the matrices become too large, the performance of R slows considerably. Rcpp, on the other hand, is capable of executing programs with a lot of matrix calculations, and other programs, in a timely manner. Unlike R, Rcpp is limited on functions, and we were forced to call R on numerous occasions to attain functions to use in our models. As a consequence of this constant reliance on R, Rcpp can perform very slowly, as was exhibited in the Probit Regression model example. Also, Rcpp's syntax is not simple as R, and there is no integrated development environment (IDE) that will clearly describe programming errors. Although Rcpp outperformed Julia several times during our research, the overall performances of Rcpp and Julia varied by very little. Julia's performance is very comparable to Rcpp's performance. Julia additionally has syntax very reminiscent of other programming languages, like MATLAB, making learning Julia an easy process. However, it is more difficult to translate types in Julia than in Rcpp and MATLAB. Since MATLAB is programmed for mathematical computations, it can perform operations, such as matrix calculations, quickly. One limitation to MATLAB is it lacks a truncated normal distribution function, forcing us to define the distribution ourselves, which slows MATLAB drastically. Furthermore, MATLAB does not handle for-loops well so MATLAB never performed faster than Rcpp or Julia.

Although we are unable to make a fair comparison of JAGS and the other four languages, JAGS is a useful program for creating Gibbs samplers, but its shortcomings may affect users seeking to run specific models with large data sets. The most beneficial aspect of JAGS is its simplicity. Rather than manually programming an entire Gibbs sampler, the user needs to only define the variables and indicate which models to use. JAGS is then able to compute the Markov chain to generate the approximate samples. However, JAGS is programmed to use the method it deems best to compute the samples, and the method chosen is ultimately unknown to the user. Furthermore, when writing scripts for JAGS, there is a significant speed performance difference between writing the script in matrix notation versus writing it *component wise*, i.e. listing the linear model row-by-row. This can cause difficulties when coding Gibbs samplers with a data set with many variables. Also, JAGS can be considerably slow in certain models, such as the linear mixed model with both the proper and improper distributions. Overall, JAGS may not be the most ideal choice when using certain statistical models.

In conclusion, each language has their own uniqueness. Each computing language makes it easier to do certain tasks than others; however, when it comes to Bayesian analysis with MCMC, the best overall language is Julia. With computing performance rivaling C++ and simple to code interface, it makes for a great tool in heavy computation.

Appendix A

One-Sample Normal Model

A.1 R/Rcpp/JAGS Workflow

```
# Set Working Directory to Source Files
getwd()
setwd()


# Call Libraries and Source Files
# install.packages("Rcpp")
# install.packages("RcppEigen")
# install.packages("coda")
# install.packages("inline")
# install.pacakges("rjags")

library(Rcpp)
library(RcppEigen)
library(coda)
library(inline)
library(rjags)

source("OneSampleNormSourceCode_2016-08-09.R") # Loads the One Sample Gibbs Sampler Functions

# Set Directory to Save Output
getwd()
setwd()

###################################
######## Simulating Dataset #######
###################################

set.seed(999) # Initialize Seed

n <- 50
outcomes <- floor(rnorm(n, mean=110, sd=13))
# the value of mu used to simulate the data is 110 and the value of sigma is 13.

write.csv(outcomes, file = "OneSampleNormalData.csv") # Saves Simulated Data for Julia

summary(outcomes)
hist(outcomes)

# Summary stats
y.bar <- mean(outcomes)  # MLE
s <- sd(outcomes)

###############################
### Finding Hyper-Parameters ###
###############################

# mu prior
find.normal(prior.mean=120, percentile=130, p=0.95)

mu.prior.mean <- 120
mu.prior.sd <- 6.08
mu.prior.precision <- 1 / mu.prior.sd^2

# plot of prior for mu
plot(density(rnorm(10000, mean=mu.prior.mean, sd=mu.prior.sd)),
     main=expression(paste("Prior Density of ", mu)),
     xlab=expression(mu), ylab="density")

# tau prior
  # Returns prior mode guess for sigma
normal.percentile.to.sd(mean.value=120, percentile=140, p=0.95)
  # Returns prior percentile guess for sigma
normal.percentile.to.sd(mean.value=120, percentile=145, p=0.95)

  # Returns shape and rate parameters for the Gamma distribution of tau
gamma.parameters <- find.tau.gamma(prior.sigma.mode=12.15,
                                   sigma.percentile=15.19, p=0.95)

tau.prior.shape <- gamma.parameters$a
tau.prior.rate <-  gamma.parameters$b

# plot of prior for tau
par(mfrow=c(1, 2))
plot(density(rgamma(10000, shape=tau.prior.shape, rate=tau.prior.rate)),
     main=expression(paste("Prior Density of ", tau)),
     xlab=expression(tau), ylab="density")
plot(density(1/sqrt(rgamma(10000, shape=tau.prior.shape, rate=tau.prior.rate ))),
     main=expression(paste("Prior Density of ", sigma)),
     xlab=expression(sigma), ylab="density")

########################################
#### Running the MCMC Gibbs sampler ###
########################################

# Set Number of Chains for Gibbs Sampler
iterations = 4

# R Function
set.seed(999)

for(l in 1:iterations){
  start.time<-Sys.time()
  MCMC <- GibbsNorm(data = outcomes, tau_prior_shape = tau.prior.shape,
                    tau_prior_rate =  tau.prior.rate,
                    mu_prior_precision = mu.prior.precision,
                    mu_prior_mean = mu.prior.mean, iterations = 500000,
                    mu_initial = 1, tau_initial = 1,
                    burnin = 500000, nthin =  1)
  Sys.time() - start.time

  print(paste(" ########################## "))
  print(paste(" This is iteration: ", l))
  print(paste(" ########################## "))

  print(summary(as.mcmc(MCMC$mu)))
  print(summary(as.mcmc(MCMC$tau)))
  print(summary(as.mcmc(MCMC$sigma)))
  #write.csv(x = MCMC, file = paste("OneSampleNormal_",l,"_iteration_R_2016-07-19.csv",sep=""))
  # Saves MCMC Chain Output
}




# Rcpp Function
set.seed(999)

for(l in 1:iterations){
  start=Sys.time()
  gibbs=GibbsNormcpp(iterations = 500000, data_sd = sd(outcomes),
                     data_mean = mean(outcomes), data_size = length(outcomes),
                     mu_prior_mean = mu.prior.mean, mu_prior_precision = mu.prior.precision,
                     tau_prior_shape = tau.prior.shape, tau_prior_rate = tau.prior.rate,
                        mu_initial = 1,tau_initial = 1, burnin = 500000, n_thin = 1)
  Sys.time()-start

  print(paste(" ########################## "))
  print(paste(" This is iteration: ", l))
  print(paste(" ########################## "))

  print(summary(as.mcmc(gibbs$Mu)))
  print(summary(as.mcmc(gibbs$Tau)))
  print(summary(as.mcmc(gibbs$Sigma)))
 # write.csv(x = MCMC, file = paste("OneSampleNormal_",l,"_iterationRcpp_2016-08-09.csv",sep
    =""))
}


# JAGS Function
set.seed(999)

for(l in 1 :iterations){
  jagsfit <- jags.model(file = "onesamplenorm.jags", #jags file
                         data = list('mu_prior_mean' = mu.prior.mean,
                                     'mu_prior_precision' = mu.prior.precision,
                                     'tau_prior_shape' = tau.prior.shape,
                                     'tau_prior_rate' = tau.prior.rate,
                                     'y' = outcomes,
                                     'N' = length(outcomes)
                         ),
                         n.chains = 1, n.adapt = 0)

  start.time <- Sys.time()
  update(jagsfit, 500000) # Progress the burn in length of the chain

  # Obtain main chain observations and monitor the parameters of interest
  MCMC.out <- coda.samples(jagsfit,
                            var = c("mu","tau"), # Tell JAGS what to keep track of
                            n.iter = 500000,
                            thin = 1)
  Sys.time()  - start.time

  print(paste(" ########################## "))
  print(paste(" This is iteration: ", l))
  print(paste(" ########################## "))

  print(summary(MCMC.out))
  #write.csv(x = MCMC, file = paste("OneSampleNormal_",l,"_iterationJAGS_2016-07-19.csv",sep
    =""))
}
```

Listing A.1: One Sample Work Flow R code

A.2 Julia

```
# Pkg.add("Distributions")
# Pkg.add("DataFrames")
using Distributions, DataFrames

srand(1234)

function GibbsNorm(iterations, burnin, nthin, mu_prior_mean, mu_prior_precision,
    tau_prior_shape, tau_prior_rate, tau_initial, mu_initial, dataset)
    n = length(dataset)
    ybar = mean(dataset)
    s = std(dataset)
    X = fill(0.0, iterations, 3) # first column for mu and second for tau, third for sigma for
     MCMC chain
    tempmu = mu_initial
    temptau = tau_initial
  post_shape = tau_prior_shape + (n / 2)
    for i in 1:burnin
    rate = tau_prior_rate + (((n-1) * s^2 + n * (ybar - tempmu)^2) / 2.0)
    temptau= rand(Gamma(post_shape, 1.0 / rate))
    w = (n * temptau) / (n * temptau + mu_prior_precision)
    tempmu= rand(Normal((w * ybar) + ((1.0 - w) * mu_prior_mean), 1.0 / sqrt(n * temptau +
     mu_prior_precision) ) )
    end

    for i in 1:iterations
      for j in 1:nthin
    rate = tau_prior_rate + (((n-1) * s^2 + n * (ybar - tempmu)^2) / 2.0)
    temptau= rand(Gamma( post_shape , 1.0 / rate))
    w = (n * temptau) / (n * temptau + mu_prior_precision)
    tempmu= rand(Normal((w * ybar) + ((1.0 - w) * mu_prior_mean), 1.0 / sqrt(n * temptau +
     mu_prior_precision) ) )
      end
      X[i, 2] = temptau
      X[i, 1] = tempmu
    X[i, 3] = 1 / sqrt(temptau) # sigma
    end

  return X
end

# Import Data
df = readtable("OneSampleNormalData.csv") # imports data and a enumerated list in first column
datas=df[:x_1] # Call data

# Set Number of Chains for Gibb Sampler
iterations = 10

for l in 1:iterations
    @time dataoutput = GibbsNorm(500000, 1, 500000, 120.0, 1/6.08^2, 21.02, 3250.647, mean(
     datas), 1.0, datas)
    describe(convert(DataFrame,dataoutput))
    # writedlm(string("OneSampleNormalProperData",l,".txt"), dataoutput )
end
```

Listing A.2: Julia Code

Appendix B

Linear Regression

B.1 R/Rcpp/JAGS Workflow

```r
# Set Working Directory to Source Files
getwd()
setwd()

# Call Libraries and Source Files
# install.packages("Rcpp")
# install.packages("RcppEigen")
# install.packages("coda")
# install.packages("inline")
# install.pacakges("rjags")
# install.packages("car")
library(Rcpp)
library(RcppEigen)
library(coda)
library(inline)
library(rjags)
library(car)

source("LinearRegression_NBADataSourceCode_2016-08-09.R") #calls the source file

# Set Directory to Save Output
getwd()
setwd()

NBA = read.csv(file = "NBA2015Data.csv",header = T)

NBA.r=subset(NBA, NBA$PTS>0 & NBA$GP>5)

fit= lm(formula = log(PTS) ~ MIN, data = NBA.r)

plot(fit$fitted.values, fit$residuals)
abline(a = 0, 0, col="red")
step(fit,direction = "backward")

ModelMatrixX = model.matrix(fit)
ModelMatrixY = log(NBA.r$PTS)
beta.hat <- solve(t(ModelMatrixX)%*%ModelMatrixX) %*% t(ModelMatrixX) %*% ModelMatrixY

write.csv(ModelMatrixX, file = "NBAmatrixX.csv")
write.csv(ModelMatrixY, file = "NBAmatrixY.csv")

prior.mean.beta <- rep(0,ncol(ModelMatrixX))
prior.cov.beta <- diag(ncol(ModelMatrixX))*100

tau.prior.shape <- 0.001
tau.prior.rate <-  0.001

########################################
#### Running the MCMC Gibbs sampler ###
########################################

```

```
# Set Number of Chains for Gibbs Sampler
iterations = 1

# R function
set.seed(999)
for (l in 1:iterations){
  start.time <- Sys.time()
 MCMC <- Gibbslm(iterations = 500000, burnin = 500000, nthin = 1,
                 prior_mean_beta = prior.mean.beta, prior_cov_beta = prior.cov.beta,
                 tau_prior_shape = tau.prior.shape, tau_prior_rate = tau.prior.rate,
                 Response = ModelMatrixY, ModelMatrixX = ModelMatrixX,
                 start.beta = beta.hat)
  print(Sys.time() - start.time)

  summary(MCMC)

  print(paste("################################# "))
  print(paste("#### This is iteration: ", l, " ####"))
  print(paste("################################# "))


  print(summary(as.mcmc(MCMC$beta)))
  print(summary(as.mcmc(MCMC$sigma)))
  # write.csv(x = MCMC,
  #           file = paste("LinearRegression_BostonDataR_",l,"_iteration_2016-07-20.csv",
  #                        sep=""))

}

# Rcpp Function
set.seed(999)
for (l in 1:iterations){
  start = Sys.time()
  Gibbs=GibbslmCpp(iterations = 500000, burnin = 500000, n_thin = 1,
                   Beta_prior_mean = prior.mean.beta, Beta_Prior_CovMat = prior.cov.beta,
                   tau_prior_shape = tau.prior.shape,
                   tau_prior_rate = tau.prior.rate,
                   Response = ModelMatrixY, ModelMatrixX = ModelMatrixX,
                   beta_initial = beta.hat)
 print (Sys.time()-start)
  summary(Gibbs)
  print(paste("###################################" ))
  print(paste("##### This is iteration: ", l, "####"))
  print(paste("###################################" ))

  for (ii in names(Gibbs)){
    print(summary(as.mcmc(Gibbs[[ii]])))
  }
  # write.csv(x = MCMC,
  #           file = paste("LinearRegression_BostonDataRcpp_",l,"_iterationRcpp_2016-07-20.csv
    ",
  #                        sep=""))
}


# JAGS Function
set.seed(999)
for(l in 1 :iterations){
  jagsfit <- jags.model(file = "LinearRegressionNBA.jags",
                        data = list('Response' = log(NBA.r$PTS),
                                    'MIN' = NBA.r$MIN,
                                    'N' = length(NBA.r$MIN),
                                    'P' = ncol(ModelMatrixX),
                                    'prior.mean' = as.vector(prior.mean.beta),
                                    'prior.precision'  = solve(prior.cov.beta),
                                    'tau_prior_shape' = tau.prior.shape,
                                    'tau_prior_rate' = tau.prior.rate),
                        inits = list('beta'= as.vector(beta.hat),'tau'=1),
                        n.chains=1,
                        n.adapt=0
  )
  start.time <- Sys.time()
  update(jagsfit, 500000) # Obtain first 100,000 (burnin draws)

  MCMC.out <- coda.samples(jagsfit,
                           var = c("beta", "sigma"),
                           n.iter = 500000,  # Obtain the main 100,000 draws
                           thin = 1)
  print(Sys.time() - start.time)

  print(paste("###################################  "))
  print(paste("#### This is iteration: ", 1, " ####"))
  print(paste("###################################  "))

  print(summary(MCMC.out))

  # write.csv(x = MCMC,
  #           file = paste("LinearRegression_NBAData",1,"_iterationJAGS_2016-07-20.csv",
  #                        sep=""))
}
```

Listing B.1: Linear Regression Work Flow R code

B.2 Julia

```
# Pkg.add("Distributions")
# Pkg.add("DataFrames")
using Distributions, DataFrames
srand(1234)

function GibbsLM(iterations, burnin, nthin, Response, ModelMatrixX, beta_prior_mean,
    beta_prior_covarmat, tau_prior_shape, tau_prior_rate, BetaInitial, TauInitial)
  n = convert(Float64, size(ModelMatrixX , 1)) # Number of rows/sample size
  m = size(ModelMatrixX, 2) # Number of columns/parameters for model
  Beta = fill(0.0, iterations, m)
  Tau = fill(0.0, iterations, 1)
  tempbeta = fill(0.0, 1, m)
  tempbeta = BetaInitial'
  temptau = TauInitial
  sigma = fill(0.0, iterations)
  BetaHat = fill(0.0,m)
  Rate = 0.0
  V_inv = fill(0.0, m , m)
  eta = fill(0.0, m)
  CholV = fill(0.0, m, m)
  tXX = ModelMatrixX'* ModelMatrixX
  tXy = ModelMatrixX' * Response
  BetaHat = inv(tXX) * transpose(ModelMatrixX) * Response
  SSE = (norm(Response - (ModelMatrixX * BetaHat)))^2
  post_shape = tau_prior_shape + (n / 2.0)

 for i in 1:burnin
    Rate =  norm((tau_prior_rate + ((SSE + (tempbeta - BetaHat') * tXX * transpose((tempbeta -
      BetaHat'))) / 2 )))
    temptau = rand(Gamma(post_shape, (1.0 / Rate) ) )
    V_inv = (temptau) * tXX + inv(beta_prior_covarmat)
    V = inv(V_inv)
    normals = rand(Normal(0,1), m)
    eta = (temptau * tXy) + (inv(beta_prior_covarmat) * beta_prior_mean)
    CholV= transpose(chol(V)) # Lower Cholosky Decomposition
    tempbeta = transpose((V * eta) + (CholV * normals))
  end

 for i in 1:iterations
   for j in 1:nthin
    Rate =  norm((tau_prior_rate + ((SSE + (tempbeta - BetaHat') * tXX * transpose((tempbeta -
      BetaHat'))) / 2 )))
    temptau = rand(Gamma(tau_prior_shape + (n / 2.0), (1.0 / Rate) ) )
    V_inv = (temptau) * tXX + inv(beta_prior_covarmat)
    V = inv(V_inv)
    normals = rand(Normal(0,1), m)
    eta = (temptau * tXy) + (inv(beta_prior_covarmat) * beta_prior_mean)
    CholV= transpose(chol(V)) # Lower Cholosky Decomposition
    tempbeta = transpose((V * eta) + (CholV * normals))
  end
```

```
    Beta[i, :] = tempbeta'
    Tau[i] = temptau
    sigma[i] = (1.0 / sqrt(Tau[i]))
  end

return [Beta Tau sigma]
end

Y = readtable("nbamatrixY.csv")
X = readtable("nbamatrixX.csv")
DatX = convert(Array{Float64,2}, X)[:, 2:end]
DatY = convert(Array{Float64,2}, Y)[:, 2:end]


prior_mean = [0.3116; 0.0766]
prior_covarmat = eye(2)*100
prior_shape = 0.001
prior_rate = 0.001


iterations = 2
for l in 1:iterations
  @time dataoutput = GibbsLM(500000, 500000, 1, DatY, DatX, prior_mean, prior_covarmat,
    prior_shape, prior_rate, [1;1], 1.0)
  describe(convert(DataFrame, dataoutput))
  # writedlm(string("LinearRegression",l,".txt"),dataoutput)
end
```
Listing B.2: Julia Code

Appendix C

Linear Mixed Models

C.1 Improper Prior – Normal Random Effects

C.1.1 Source code

```r
###################
### Source File ###
###################

###########################
######## R Function #######
###########################

GibbslmeImproper<- function(iterations, burnin, nthin = 1,
                            Response, ModelMatrixX, ModelMatrixZ,
                            tau_prior_shape = c(0, -1/2),
                            tau_prior_rate = c(0,0), start.theta){
  N <- length(Response)
  p <- ncol(ModelMatrixX)
  q <- ncol(ModelMatrixZ)
  W <- cbind(ModelMatrixX, ModelMatrixZ)
  a_pos <- c( tau_prior_shape[1] + N / 2.0, tau_prior_shape[2] + q / 2.0)

  tXX <- t(ModelMatrixX) %*% ModelMatrixX
  tXZ <- t(ModelMatrixX) %*% ModelMatrixZ
  tZX <- t(ModelMatrixZ) %*% ModelMatrixX
  tZZ <- t(ModelMatrixZ) %*% ModelMatrixZ
  tXy <- t(ModelMatrixX) %*% Response
  tZy <- t(ModelMatrixZ) %*% Response

  thetas <- matrix(0, nrow = iterations, ncol = {p+q})
  lambdas <- matrix(0, nrow = iterations, ncol = 2)
  temp_thetas <- start.theta
  temp_lambdas <- c(0,0)
  I.q <- diag(q)
  eta <- rep(0, p + q)
  V_inv <- matrix(0 , nrow = p + q, ncol = p + q)

  for(j in 1:burnin){
    diff <- Response - W %*% temp_thetas
    temp_lambdas[1] <- rgamma(1, shape = a_pos[1], rate = tau_prior_rate[1] + t(diff) %*% diff
      / 2.0)
    diffu <- temp_thetas[{p+1}:{p+q}]
    temp_lambdas[2] <- rgamma(1, shape = a_pos[2], rate = tau_prior_rate[2] + t(diffu) %*%
      diffu / 2.0)

    V_inv[1:p, 1:p] <- temp_lambdas[1] * tXX
    V_inv[1:p, {p+1}:{p+q}] <- temp_lambdas[1] * tXZ
    V_inv[{p+1}:{p+q}, 1:p] <- temp_lambdas[1] * tZX
    V_inv[{p+1}:{p+q}, {p+1}:{p+q}] <- temp_lambdas[1] * tZZ + temp_lambdas[2] * I.q
    V <- chol2inv(chol(V_inv))
    eta[1:p] <- temp_lambdas[1] * tXy
    eta[{p+1}:{p+q}] <- temp_lambdas[1] * tZy
    temp_thetas <- V %*% eta + t(chol(V)) %*% rnorm(p+q)
  }

  for( i in 1:iterations){
    for(j in 1:nthin){
      diff <- Response - W %*% temp_thetas
      temp_lambdas[1] <- rgamma(1, shape = a_pos[1], rate = tau_prior_rate[1] + t(diff) %*%
    diff / 2.0)
      diffu = temp_thetas[{p+1}:{p+q}]
      temp_lambdas[2] <- rgamma(1, shape = a_pos[2], rate = tau_prior_rate[2] + t(diffu) %*%
    diffu / 2.0)

      V_inv[1:p, 1:p] <- temp_lambdas[1] * tXX
      V_inv[1:p, {p+1}:{p+q}] <- temp_lambdas[1] * tXZ
      V_inv[{p+1}:{p+q}, 1:p] <- temp_lambdas[1] * tZX
      V_inv[{p+1}:{p+q}, {p+1}:{p+q}] <- temp_lambdas[1] * tZZ + temp_lambdas[2] * I.q
      V <- chol2inv(chol(V_inv))
      eta[1:p] <- temp_lambdas[1] * tXy
      eta[{p+1}:{p+q}] <- temp_lambdas[1] * tZy
      temp_thetas <- V %*% eta + t(chol(V)) %*% rnorm(p+q)
    }
    thetas[i , ] <- temp_thetas
    lambdas[i , ] <- temp_lambdas
  }
  sigmas <- 1 / sqrt(lambdas)
  thetas <- thetas
  sigmas <- sigmas
  return( list( beta = thetas[ , 1:p], group = thetas[ , {p+1}:{p+q}], sigma = sigmas) )
}

##############################
######## Rcpp Function #######
##############################

src_eigen_imp<- '
using Eigen::Map ;
using Eigen::MatrixXd ;
using Eigen::VectorXd ;
using Eigen::Vector2d ;
using Rcpp::as ;

typedef Eigen::Map<Eigen::MatrixXd> MapMatd ;
typedef Eigen::Map<Eigen::VectorXd> MapVecd ;

int net_iterations = Rcpp::as<int>(iterations);
int burn = Rcpp::as<int>(burnin);
int n_thin = Rcpp::as<int>(nthin);

Rcpp::NumericMatrix Xc(ModelMatrixX) ;
Rcpp::NumericMatrix Zc(ModelMatrixZ) ;
Rcpp::NumericVector yc(Response) ;

const MapMatd    X(Rcpp::as<MapMatd>(Xc)) ;
const MapMatd    Z(Rcpp::as<MapMatd>(Zc)) ;
const MapVecd    y(Rcpp::as<MapVecd>(yc)) ;

int N = X.rows(), p = X.cols(), q = Z.cols() ;

Rcpp::NumericVector starttheтac(starttheta) ;
const MapVecd     start_theta(Rcpp::as<MapVecd>(startthetac)) ;

Rcpp::NumericVector ac(tau_prior_shape) ;
Rcpp::NumericVector bc(tau_prior_rate) ;

const MapVecd    a(Rcpp::as<MapVecd>(ac));
const MapVecd    b(Rcpp::as<MapVecd>(bc));
VectorXd   a_pos = a ;
a_pos[0] = a[0] + N * 0.5 ;
a_pos[1] = a[1] + q * 0.5 ;

const MatrixXd tXX = X.transpose() * X ;
const MatrixXd tXZ = X.transpose() * Z ;
const MatrixXd tZX = Z.transpose() * X ;
const MatrixXd tZZ = Z.transpose() * Z ;
```

```
 const VectorXd tXy = X.transpose() * y ;
 const VectorXd tZy = Z.transpose() * y ;

 MatrixXd   W(N, p+q);
W.leftCols(p) = X ;
W.rightCols(q) = Z ;

 MatrixXd thetas(net_iterations , p+q) ;
 MatrixXd sigmas(net_iterations , 2) ;
 VectorXd temp_thetas = start_theta;
 VectorXd temp_lambdas(2); temp_lambdas << 0, 0;
 const MatrixXd identity_q = MatrixXd::Identity(q, q);
 const MatrixXd identity_pq = MatrixXd::Identity(p+q, p+q);

 VectorXd eta(p+q) ;
 MatrixXd V_inv(p+q, p+q) ;
 MatrixXd V(p+q, p+q);
 VectorXd diff = y - W * temp_thetas ;

RNGScope scp;
Rcpp::Function rnorm(”rnorm”) ;
Rcpp::Function rgamma(”rgamma”) ;
MapVecd normals = Rcpp::as<MapVecd>(Rcpp::rnorm(p+q)) ;

 for(int j = 0; j < burn; j++){
   diff = y - W * temp_thetas ;
   temp_lambdas[0] = Rcpp::as<double>(Rcpp::rgamma(1, a_pos[0],
                                      1.0 / (b[0] + 0.5 * diff.squaredNorm()))) ;
   temp_lambdas[1] = Rcpp::as<double>(Rcpp::rgamma(1, a_pos[1],
                                      1.0/ (b[1] + 0.5 * temp_thetas.tail(q).squaredNorm()))) ;
   V_inv.topLeftCorner(p, p) = temp_lambdas[0] * tXX  ;
   V_inv.topRightCorner(p, q) = temp_lambdas[0] * tXZ ;
   V_inv.bottomLeftCorner(q, p) = temp_lambdas[0] * tZX ;
   V_inv.bottomRightCorner(q, q) = temp_lambdas[0] * tZZ + temp_lambdas[1] * identity_q ;

   V = V_inv.inverse();
   eta.head(p) =  temp_lambdas[0] * tXy ;
   eta.tail(q) =  temp_lambdas[0] * tZy ;
   normals = Rcpp::as<MapVecd>(Rcpp::rnorm(p+q)) ;
   temp_thetas = V * eta + V.llt().matrixL() * normals ;
 }

 for(int i = 0; i < net_iterations; i++){
   for(int j = 0; j < n_thin; j++){
     diff = y - W * temp_thetas ;
     temp_lambdas[0] = Rcpp::as<double>(Rcpp::rgamma(1, a_pos[0],
                                        1.0 / (b[0] + 0.5 * diff.squaredNorm()))) ;
     temp_lambdas[1] = Rcpp::as<double>(Rcpp::rgamma(1, a_pos[1],
                                        1.0/ (b[1] + 0.5 * temp_thetas.tail(q).squaredNorm()))) 
     ;
     V_inv.topLeftCorner(p, p) = temp_lambdas[0] * tXX  ;
     V_inv.topRightCorner(p, q) = temp_lambdas[0] * tXZ ;
     V_inv.bottomLeftCorner(q, p) = temp_lambdas[0] * tZX ;
     V_inv.bottomRightCorner(q, q) = temp_lambdas[0] * tZZ + temp_lambdas[1] * identity_q ;

     V = V_inv.inverse();
     eta.head(p) =  temp_lambdas[0] * tXy ;
     eta.tail(q) =  temp_lambdas[0] * tZy ;
     normals = Rcpp::as<MapVecd>(Rcpp::rnorm(p+q)) ;
     temp_thetas = V * eta + V.llt().matrixL() * normals ;
   }
 thetas.row(i) = temp_thetas ;
 sigmas.row(i) = 1 / temp_lambdas.array().sqrt() ;
 }

 MatrixXd  betas = thetas.leftCols(p);
 MatrixXd  us = thetas.rightCols(q);

 return Rcpp::List::create(
Rcpp::Named(”beta”) = betas,
Rcpp::Named(”group”) = us,
Rcpp::Named(”sigma”) = sigmas);
 ’
```

```

GibbslmeImproperCpp <- cxxfunction(signature(iterations = "int", nthin = "int", burnin = "int"
    ,
                                              Response = "numeric",
                                              ModelMatrixX = "numeric",
                                              ModelMatrixZ = "numeric",
                                              tau_prior_shape = "numeric",
                                              tau_prior_rate = "numeric",
                                              starttheta = "numeric"),
                                              src_eigen_imp, plugin="RcppEigen")

############################################
#### JAGS ###

cat( "
     var
     Response[N], Beta[P], MIN[N], u[q], cutoff[q+1],
     prior.mean[P], prior.precision[P, P], mu[N],
     tau_prior_shape[2], tau_prior_rate[2], tau_e, tau_u, tau[N];
     model{
     # Likelihood specification
     for(i in 1:q){
     for(k in (cutoff[i]+1):cutoff[i+1]){
     Response[k] ~ dnorm(mu[k], tau[k])
     mu[k] <- Beta[1] + Beta[2] * MIN[k] + u[i]
     tau[k] <- 1/((1 / tau_e) + (1 / tau_u))
     }
     u[i] ~ dnorm(0, tau_u)
     }

     # Prior specification
     Beta[] ~ dmnorm(prior.mean[], prior.precision[,])

     tau_u ~ dgamma(tau_prior_shape[1], tau_prior_rate[1])
     tau_e ~ dgamma(tau_prior_shape[2], tau_prior_rate[2])
     sigma_e <- sqrt(1 / tau_e)
     sigma_u <- sqrt(1 / tau_u)
     }",
     file="LMM_nba.jags")
```
Listing C.1: Linear Mixed Model with Improper Priors R Source Code

GibbslmeImproper input description:

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `ModelMatrixZ`: A sparse matrix filled with 0's and 1's that associates each observation to a random effect
- `tau_prior_shape`: A numeric vector that provides the shape parameter for the prior distribution of τ_e and τ_u respectively
- `tau_prior_rate`: A numeric vector that provides the rate parameter for the prior distribution of τ_e and τ_u respectively
- `start.theta`: A concatenated numeric vector of initial values for MCMC of β and u

GibbslmeImproperCpp input description:

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `ModelMatrixZ`: A sparse matrix filled with 0's and 1's that associates each observation to a random effect
- `tau_prior_shape`: A numeric vector that provides the shape parameter for the prior distribution of τ_e and τ_u respectively
- `tau_prior_rate`: A numeric vector that provides the rate parameter for the prior distribution of τ_e and τ_u respectively
- `start.theta`: A concatenated numeric vector of initial values for MCMC of β and u

C.1.2 R/Rcpp/JAGS Workflow

```r
getwd()
setwd()

#call libraires and source
# install.packages("nlme")
# install.packages("Rcpp")
# install.packages("RcppEigen")
# install.packages("coda")
# install.packages("inline")
# install.packages("rjags")


library(nlme)
library(Rcpp)
library(RcppEigen)
library(coda)
library(inline)
library(rjags)

source("LinearMixedModel_Improper_NBADataSourceCode_2016-08-09.R") #calls the source f

getwd()
setwd()

nba <- read.csv(file="NBA2015Data.csv", header=TRUE)

# We won't use attach(nba) in this example

plot(nba$MIN, nba$PTS, xlab="minutes", ylab="points per game")



#####################################
#      Frequentist analysis        #
#####################################

# Let's model log(PTS) vs MIN; log = natural log
# But now we treat the TEAM variable as a random effect

# Be careful a few players have PTS=0
which(nba$PTS==0)

# Let's look at their data
```

```
nba[which(nba$PTS==0), ]

#  Let's remove some problematic observations from the data set
nba.r=subset(nba, nba$PTS>0 & nba$GP>5)
nba.r <- nba.r[order(nba.r$TEAM), ]
# sort data by team ; this is very important for our "home-made" Gibbs sampler.

nba.r$log.PTS <- log(nba.r$PTS)

dim(nba)
dim(nba.r)  # 6 players have been removed

team.size <- as.numeric(table(nba.r$TEAM))
cutoff <- cumsum(c(0,team.size))


# Consider the following random intercept model
log.fit.mixed <- lme( log.PTS ~ MIN, random = ~1 | TEAM, data = nba.r)
summary(log.fit.mixed)
coefficients(log.fit.mixed) # beta_1 + u_j
log.fit.mixed$coeff$random # u_j
intervals(log.fit.mixed)

library(lattice)

xyplot(log.PTS ~ MIN | TEAM, groups=TEAM, type=c("p", "r"), data=nba.r)

# Predict PPG for a San Antonio Spurs player that plays 25 minutes
new <- data.frame(MIN=25, TEAM="SAS")
exp(predict(log.fit.mixed, newdata=new))

# Compare old model with the new mixed model
log.fit <- lm(log(PTS) ~ MIN, data=nba.r)
plot(nba.r$MIN, nba.r$PTS, xlab="minutes", ylab="points per game")

original.fit <- function(x){
  y.hat <- coef(log.fit)[1] + coef(log.fit)[2] * x
  return(exp(y.hat))
}
x <- seq(from=min(nba.r$MIN), to=max(nba.r$MIN), by=0.01)  # Create sequence of points
lines(x, original.fit(x), col="red")

pred.PTS <- exp(fitted(log.fit.mixed, newdata=new, level=0:1))
points(nba.r$MIN, pred.PTS[ , 2], col="red", pch=3)

# standardized residuals versus fitted values by TEAM
plot(log.fit.mixed, resid(., type = "p") ~ fitted(.) | TEAM, abline = 0)
# box-plots of residuals by TEAM
plot(log.fit.mixed, TEAM ~ resid(.))
# observed versus fitted values by TEAM
plot(log.fit.mixed, log.PTS ~ fitted(.) | TEAM, abline = c(0, 1))


########################################
#  Bayesian Analysis Reference Prior  #
########################################
log.fit.mixed <- lme( log.PTS ~ MIN, random = ~1 | TEAM, data = nba.r)

# Make sure that the data are sorted by TEAM
ModelMatrixY <- log(nba.r$PTS)

log.fit.fixed <- lm(log(PTS) ~ MIN, data=nba.r)
ModelMatrixX <- model.matrix(log.fit.fixed) # trick to get the X matrix

log.fit.random <- lm(log(PTS) ~ TEAM - 1, data=nba.r)
ModelMatrixZ <- model.matrix(log.fit.random) #  trick to get Z matrix

beta.hat <- as.vector(log.fit.mixed$coeff$fixed)
u.hat <- coefficients(log.fit.mixed)[ , 1] - as.vector(log.fit.mixed$coeff$fixed)[1]
start.thetas <- c(beta.hat, u.hat)

prior.mean.beta = rep(0.0, ncol(ModelMatrixX))
```

```r
prior.cov.beta = diag(ncol(ModelMatrixX)) * 100
tau.prior.rate = 1
tau.prior.shape = 1

beta.hat <- solve(t(ModelMatrixX)%*%ModelMatrixX) %*% t(ModelMatrixX) %*% ModelMatrixY

iterations = 1

### R Code ###
set.seed(999)

for(l in 1:iterations){
  start.time<-Sys.time()
 MCMC <- GibbslmeImproper(iterations = 500000, nthin = 1, burnin = 500000,
                          start.theta = start.thetas, ModelMatrixX = ModelMatrixX,
                          ModelMatrixZ = ModelMatrixZ, Response = ModelMatrixY)
  # by default a=c(0, -0.5), b=c(0,0)
  print(Sys.time() - start.time)

  print("###################################################")
  print("###################################################")
  print(paste("############ This is iteration: ", l,"##############"))
  print("###################################################")
  print("###################################################")

  for(r in names(MCMC)){
    print(summary(as.mcmc(MCMC[[r]])))
  }
#    write.csv(x = MCMC,
#              file = paste("LinearMixedModelNBAData_",l,"_iterationR_2016-07-20.csv",
#                           sep=""))
# }

### Rcpp Code ###

set.seed(999)
for(l in 1:iterations){
  start.time<-Sys.time()
 MCMC <- GibbslmeImproperCpp(iterations=500000, nthin=1, burnin=500000,
                             starttheta=start.thetas,ModelMatrixX = ModelMatrixX,
                             ModelMatrixZ = ModelMatrixZ, Response = ModelMatrixY,
                             tau_prior_shape = c(0, -0.5), tau_prior_rate = c(0,0))
  Sys.time() - start.time

  print("###################################################")
  print("###################################################")
  print(paste("############ This is iteration: ", l,"##############"))
  print("###################################################")
  print("###################################################")

  for(r in names(MCMC)){
    print(summary(as.mcmc(MCMC[[r]])))
  }
  # write.csv(x = MCMC,
  #           file = paste("LinearMixedModelNBAData_",l,"_iterationRcpp_2016-07-20.csv",
  #                        sep=""))
}


### JAGS Code ###

prior.cov.beta = diag(ncol(ModelMatrixX)) * 1000000
set.seed(999)
for(l in 1:iterations){
  set.seed(999)
  jagsfit <- jags.model(file = "LMM_nba.jags",
                        data = list('Response' = ModelMatrixY,
                                    'MIN' = nba.r$MIN,
                                    'cutoff' = cutoff,
                                    'N' = length(ModelMatrixY),
                                    'P' = ncol(ModelMatrixX),
                                    'q' = ncol(ModelMatrixZ),
                                    'prior.mean' = as.vector(prior.mean.beta),
                                    'prior.precision' = solve(prior.cov.beta),
                                    'tau_prior_shape' = c(0.001, 0.001),
                                    'tau_prior_rate' = c(0.001, 0.001)),
                        inits = list('Beta'= as.vector(beta.hat),'tau_e' = 1, 'tau_u' = 1,
                                     'u' = u.hat),
                        n.chains=1,
                        n.adapt=0
  )

  start.time <- Sys.time()
  update(jagsfit, 500000) # Obtain first 100,000 (burnin draws)

 MCMC.out <- coda.samples(jagsfit,
                           var = c("Beta", "u", "sigma_e", "sigma_u"),
                           n.iter = 500000,  # Obtain the main 100,000 draws
                           thin = 1)
  print(Sys.time() - start.time)

  # write.csv(x = as.mcmc(MCMC.out),
  #          file = paste("LinearMixedModelNBAData_multiple_length_",
  #                         1,"_iterationJAGS_2016-08-03.csv",sep=""))

  # print("#####################################################")
  # print("#####################################################")
  # print(paste("############ This is iteration: ", 1,"##############"))
  # print("#####################################################")
  # print("#####################################################")
  #
  # print(summary(MCMC.out)) # Notice the iterations being used
}
```

Listing C.2: Linear Mixed Model with Improper Priors R Work Flow

jagsfit input description:

- `Response`: A numeric vector of observed data for linear model
- `MIN`: A numeric vector of observed data for the variable MIN for linear model of NBA
- `cutoff`: A numeric vector of cumulatively summed entries of the number of players in each team of the NBA 2015 season data used for the random effect
- `N`: Sample size of the observed values
- `P`: The number of columns for the model matrix of linear model, i.e. (Number of predictors used for linear model) + 1
- `q`: The number of teams considered for the model based on the data set
- `prior.mean`: A numeric vector for the mean parameter of the normal distribution of β
- `prior.precision`: A numeric matrix for the covariance matrix parameter of the normal distribution of β
- `tau_prior_shape`: A numeric value that provides the shape parameter for the prior distribution of τ
- `tau_prior_rate`: A numeric value that provides the rate parameter for the prior distribution of τ
- `Beta`: A numeric vector of initial values for MCMC for β
- `tau_e`: A numeric value for initializing MCMC for τ_e
- `tau_u`: A numeric value for initializing MCMC for τ_u
- `u`: A numeric vector of initial values for MCMC of u

C.1.3 Julia

```
using Distributions, DataFrames
srand(1234)
function GibbsLMEimproper(iterations,burnin, nthin, Response, ModelMatrixX, ModelMatrixZ,
    tau_prior_shape, tau_prior_rate, starttheta)
  X = ModelMatrixX
  Y = Response
  Z = ModelMatrixZ
  a = tau_prior_shape
  b = tau_prior_rate
  N = size(X,1) #number of rows of X
  p = size(X,2) #number of columns of X
  q = size(Z,2) #number of columns of Z
  apos = a
  apos[1] = a[1] + N * 0.5
  apos[2] = a[2] + q * 0.5
  tXX = X' * X
  tXZ = X' * Z
  tZX = Z' * X
  tZZ = Z' * Z
  tXY = X' * Y
  tZY = Z' * Y
  W = [X Z]

  thetas = fill(0.0, iterations, p + q) #storage for MCMC theta
  sigmas = fill(0.0, iterations, 2) #storage for MCMC sigmas
  temptheta = fill(1.0, 1,p+q)
  temptheta = starttheta'

  templambda = fill(0.0,1,2)
  identityq = eye(q)
  identitypq = eye(p+q)
  diff = Y - W * temptheta'

  sigmas = fill(0.0, iter, 2)
  VinvTL = fill(0.0, p, p)
  VinvTR = fill(0.0, p, q)
  VinvBL = fill(0.0, q, p)
  VinvBR = fill(0.0, q, q)
  Vinv = fill(0.0, p+q, p+q)
  V = fill(0.0, p+q, p+q)
  etaheadp = fill(0.0, 1, p)
  etatailq = fill(0.0, 1, q)
  eta = fill(0.0, 1, p+q)
  Vchol = fill(0.0, p+q,p+q)
  Vcholup = fill(0.0, p+q,p+q)
  stdnormal = fill(0.0, 1, p+q)
  Term1 = fill(0.0,  p+q, 1)
  sigma = fill(0.0,1,2)
  Term2 = fill(0.0,  p+q, 1)

  for j in 1:burnin
    diff = Y - W * temptheta'
        templambda[1] = rand(Gamma(apos[1], 1.0/ (b[1] + 0.5 * norm(diff)^2 ) ) )
        templambda[2] = rand(Gamma(apos[2], 1.0/ (b[2] + 0.5 * norm(transpose(temptheta[(p+1)
    :(p+q)]) )^2 ) ) )
        sigma[1] = 1.0/sqrt(templambda[1])
        sigma[2] = 1.0/sqrt(templambda[2])
        VinvTL = templambda[1] * tXX
        VinvTR = templambda[1] * tXZ
        VinvBL = templambda[1] * tZX
        VinvBR = (templambda[1] * tZZ) + (templambda[2] * identityq)

        Vinv=[VinvTL VinvTR; VinvBL VinvBR]
        V = inv(Vinv)
        etaheadp = (templambda[1] * tXY)
        etatailq = (templambda[1] * tZY)
        eta = [etaheadp' etatailq']

        Vcholup = chol(V)
        Vchol = Vcholup'
        stdnormal = rand(Normal(0,1), p+q)
        Term1 = (V * eta')
        Term2 = reshape(Vchol * stdnormal, p+q,1)
        temptheta = transpose(Term1 + Term2)
      end

  for i in 1:iterations
      for j in 1:nthin
          diff = Y - W * temptheta'
          templambda[1] = rand(Gamma(apos[1], 1.0/ (b[1] + 0.5 * norm(diff)^2 ) ) )
          templambda[2] = rand(Gamma(apos[2], 1.0/ (b[2] + 0.5 * norm(transpose(temptheta[(p
    +1):(p+q)]) )^2 ) ) )
          sigma[1] = 1.0/sqrt(templambda[1])
          sigma[2] = 1.0/sqrt(templambda[2])

      VinvTL = templambda[1] * tXX
          VinvTR = templambda[1] * tXZ
          VinvBL = templambda[1] * tZX
          VinvBR = (templambda[1] * tZZ) + (templambda[2] * identityq)

          Vinv=[VinvTL VinvTR; VinvBL VinvBR]
          V = inv(Vinv)
          etaheadp = (templambda[1] * tXY)
          etatailq = (templambda[1] * tZY)
          eta = [etaheadp' etatailq']

          Vcholup = chol(V)
          Vchol = Vcholup'
          stdnormal = rand(Normal(0,1), p+q)
          Term1 = (V * eta')
          Term2 = reshape(Vchol * stdnormal, p+q,1)
          temptheta = transpose(Term1 + Term2)
      end
      thetas[i,:] = temptheta
      sigmas[i,:] = sigma
  end
  BetaMCMC = thetas[:, 1:p]
  UMCMC = thetas[:, (p+1):(p+q)]

  return [BetaMCMC UMCMC sigmas]
end

Y = readtable("matrixy.csv")
X = readtable("matrixX.csv")
Z = readtable("matrixz.csv")
initialtheta = readtable("initialization.csv")
DatX = convert(Array{Float64,2}, X)[:,2:end]
DatY = convert(Array{Float64,2}, Y)[:,2:end]
DatZ = convert(Array{Float64,2}, Z)[:,2:end]
thetastart = convert(Array{Float64,2}, initialtheta)


iterations = 10
for(l in 1:iterations)
  @time dataoutput = GibbsLMEimproper(500000, 500000, 1, DatY, DatX, DatZ, [0 -0.5], [0 0],
    thetastart)
    describe(convert(DataFrame, dataoutput))
    writedlm(string("LinearRegression_NBAData_",l,".txt"), dataoutput )
end
```

Listing C.3: Linear Mixed Model Julia code

GibbsLMEimproper input description:

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model

- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `ModelMatrixZ`: A sparse matrix filled with 0's and 1's that associates each observation to a random effect
- `tau_prior_shape`: A numeric vector that provides the shape parameter for the prior distribution of τ_e and τ_u respectively
- `tau_prior_rate`: A numeric vector that provides the rate parameter for the prior distribution of τ_e and τ_u respectively
- `starttheta`: A concatenated numeric vector of initial values for MCMC of β and u

C.2 Proper Prior – Normal Random Effects

C.2.1 Source code

```r
# Source Code for LMM with proper priors and normal random effects

##################
### R Function ###
##################

GibbsLMM = function(iterations, burnin, nthin, Response, ModelMatrixX, ModelMatrixZ,
                    prior_mean_beta, prior_cov_beta, prior_gamma_shape,
                    prior_gamma_rate, start_theta){

  X <- ModelMatrixX
  y <- Response
  Z <- ModelMatrixZ
  Sigma_beta <- prior_cov_beta
  Sigma_beta_inv <- solve(prior_cov_beta)
  mu_beta <- prior_mean_beta
  a <- prior_gamma_shape # Shape for e and u
  b <- prior_gamma_rate # Rate for e and u


  N <- length(y) # sample size
  p <- ncol(X) # number of columns of X
  q <- ncol(Z) # number of columns of Z
  W <- cbind(X, Z)
  apos = c(a[1] + N * 0.5, a[2] + q * 0.5)
  tXX <- t(X) %*% X
  tXZ <- t(X) %*% Z
  tZX <- t(Z) %*% X
  tZZ <- t(Z) %*% Z
  tXy <- t(X) %*% y
  tZy <- t(Z) %*% y

  thetas <- matrix(NA, nrow = iterations, ncol = {p+q})
  lambdas <- matrix(NA, nrow = iterations, ncol = 2)
  temp_thetas = start_theta # (beta, u)
  temp_lambdas = c(0,0) #(lambda_e, lambda_u)
  eta = rep(0, q)
  V_inv = matrix(NA, nrow = p + q, ncol = p + q)
  D_eta = diag(q)
  for (j in 1 : burnin){
    test = y- (W %*% temp_thetas)
    Fnorm = norm(x = test, type="F")
    temp_lambdas[1] = rgamma(1, apos[1], b[1] + (Fnorm^2) * 0.5)
    SecondFnorm = norm(x = D_eta %*% temp_thetas[(p+1):(p+q)], type = "F")^2
    temp_lambdas[2] = rgamma(1, apos[2], b[2] + SecondFnorm*0.5)

    V_inv[1:p, 1:p] <- temp_lambdas[1] * tXX + Sigma_beta_inv
    V_inv[1:p, {p+1}:{p+q}] <-  temp_lambdas[1] * tXZ
    V_inv[{p+1}:{p+q}, 1:p] <-  temp_lambdas[1] * tZX
    V_inv[{p+1}:{p+q}, {p+1}:{p+q}] <-  temp_lambdas[1] * tZZ + temp_lambdas[2] * D_eta

    V <- chol2inv(chol(V_inv))

    NextTerm1 <- temp_lambdas[1] * tXy + Sigma_beta_inv %*% mu_beta
    NextTerm2 <- temp_lambdas[1] * tZy

    zeta <- c(NextTerm1, NextTerm2)

    Vchol <- t(chol(V)) # cholesky decomposition
    temp_thetas <- V %*% zeta + Vchol %*% rnorm(p+q)

  }

  for(i in 1 : iterations){
    for (j in 1 : nthin){
      test= y- (W %*% temp_thetas)
      Fnorm = norm(x = test, type="F")
      temp_lambdas[1] = rgamma(1, a[1] + N * 0.5, b[1] + (Fnorm^2) * 0.5)
      SecondFnorm = norm(x = D_eta %*% temp_thetas[(p+1):(p+q)], type = "F")^2
      temp_lambdas[2] = rgamma(1, a[2] + q * 0.5, b[2] + SecondFnorm*0.5)

      V_inv[1:p, 1:p] <-  temp_lambdas[1] * tXX + Sigma_beta_inv
      V_inv[1:p, {p+1}:{p+q}] <-  temp_lambdas[1] * tXZ
      V_inv[{p+1}:{p+q}, 1:p] <-  temp_lambdas[1] * tZX
      V_inv[{p+1}:{p+q}, {p+1}:{p+q}] <-  temp_lambdas[1] * tZZ + temp_lambdas[2] * D_eta

      V <- chol2inv(chol(V_inv))

      NextTerm1 <- temp_lambdas[1] * tXy + Sigma_beta_inv %*% mu_beta
      NextTerm2 <- temp_lambdas[1] * tZy

      zeta <- c(NextTerm1, NextTerm2)

      Vchol <- t(chol(V)) # cholesky decomposition
      temp_thetas <- V %*% zeta + Vchol %*% rnorm(p+q)
    }
    thetas[i , ] <- temp_thetas
    lambdas[i , ] <- temp_lambdas
  }

  sigmas <- 1 / sqrt(lambdas)

  return( list( beta = thetas[, 1:p], group = thetas[, {p+1}: {p+q}], sigma = sigmas) )
}

#####################
### Rcpp Function ###
#####################

src_eigen_imp<- '

using Eigen :: Map ;
using Eigen :: MatrixXd ;
using Eigen :: VectorXd ;
using Eigen :: Vector2d ;
using Rcpp :: as ;

typedef Eigen :: Map<Eigen::MatrixXd> MapMatd ;
typedef Eigen :: Map<Eigen::VectorXd> MapVecd ;

int MCMCiter = Rcpp::as<int>(iterations);
int burnin = Rcpp :: as<int>(Burnin);
int n_thin = Rcpp :: as<int>(nthin);

Rcpp :: NumericMatrix Xc(ModelMatrixX) ;
Rcpp :: NumericMatrix Zc(ModelMatrixZ) ;
Rcpp :: NumericMatrix Sigma_betac(prior_cov_beta) ;
Rcpp :: NumericVector yc(Response) ;
Rcpp :: NumericVector mu_betac(prior_mean_beta) ;

const MapMatd X(Rcpp :: as<MapMatd>(Xc)) ;
const MapMatd Z(Rcpp :: as<MapMatd>(Zc)) ;
const MapMatd Sigma_beta(Rcpp :: as<MapMatd>(Sigma_betac)) ;
const MapVecd y(Rcpp :: as<MapVecd>(yc)) ;
const MapVecd mu_beta(Rcpp :: as<MapVecd>(mu_betac)) ;

const MatrixXd Sigma_beta_inv = Sigma_beta.inverse() ;

int N = y.rows(), p = X.cols(), q = Z.cols() ;

Rcpp :: NumericVector starttheta c(starttheta) ;
const MapVecd     start_theta(Rcpp::as<MapVecd>(startthetac)) ;

Rcpp :: NumericVector ac(prior_gamma_shape) ;
Rcpp :: NumericVector bc(prior_gamma_rate) ;

const MapVecd a(Rcpp::as<MapVecd>(ac)) ;
const MapVecd b(Rcpp::as<MapVecd>(bc)) ;

const MatrixXd tXX = X.transpose() * X ;
const MatrixXd tXZ = X.transpose() * Z ;
const MatrixXd tZX = Z.transpose() * X ;
const MatrixXd tZZ = Z.transpose() * Z ;
const VectorXd tXy = X.transpose() * y ;
const VectorXd tZy = Z.transpose() * y ;

MatrixXd thetas(MCMCiter, p+q) ;
MatrixXd sigmas(MCMCiter, 2) ;
thetas.col(0) = start_theta ;
VectorXd temp_thetas = start_theta ;
VectorXd temp_lambdas(2); temp_lambdas << 0, 0 ;

VectorXd zeta(p+q) ;
MatrixXd T_lambda(p,p) ;
MatrixXd V_inv(p+q, p+q) ;
MatrixXd V(p+q, p+q) ;

const MatrixXd identity_N = MatrixXd :: Identity(N,N) ;
const MatrixXd identity_q = MatrixXd :: Identity(q,q) ;
MatrixXd test(N,1) ;

MatrixXd W(N, p+q) ;
W.leftCols(p) = X ;
W.rightCols(q) = Z ;

double FNorm ;
double SecondFNorm ;

RNGScope scp ;
Rcpp :: Function rnorm("rnorm") ;
Rcpp :: Function rgamma("rgamma") ;
Rcpp :: Function fnorm("frobenius.norm") ;

MapVecd normals = Rcpp::as<MapVecd>(Rcpp::rnorm(p+q)) ;

for(int j = 0; j < burnin; j++){

test = y − (W * temp_thetas) ;

FNorm = Rcpp :: as<double>(fnorm(test)) ;

temp_lambdas[0] = Rcpp :: as<double>(Rcpp :: rgamma(1,
                     a[0] + N * 0.5 , 1.0 / (b[0] + pow(FNorm , 2.0 ) * 0.5 ) ) ) ;

SecondFNorm = temp_thetas.tail(q).transpose() * temp_thetas.tail(q) ;

temp_lambdas[1] = Rcpp :: as<double>(Rcpp :: rgamma(1,
                  a[1] + q * 0.5 , 1.0 / ( b[1]  + SecondFNorm * 0.5 ) ) )     ;

V_inv.topLeftCorner(p, p) =temp_lambdas[0] * tXX + Sigma_beta_inv ;
V_inv.topRightCorner(p, q) = temp_lambdas[0] * tXZ ;
V_inv.bottomLeftCorner(q, p) = temp_lambdas[0] * tZX ;
V_inv.bottomRightCorner(q, q) = temp_lambdas[0] * tZZ + temp_lambdas[1] * identity_q ;
V = V_inv.inverse() ;

zeta.head(p) = temp_lambdas[0] * tXy + Sigma_beta_inv * mu_beta ;
zeta.tail(q) = temp_lambdas[0] * tZy ;
```

```
normals = Rcpp::as<MapVecd>(Rcpp::rnorm(p+q)) ;

temp_thetas = V * zeta + V.llt().matrixL() * normals ;

}

for(int i = 0; i < MCMCiter; i++){
for(int j = 0; j < n_thin; j++){

test = y - (W * temp_thetas) ;

FNorm = Rcpp :: as<double>(fnorm(test)) ;

temp_lambdas[0] = Rcpp :: as<double>(Rcpp :: rgamma(1,
                        a[0] + N * 0.5 , 1.0 / (b[0] + pow(FNorm , 2.0 ) * 0.5 ) ) ) ;

SecondFNorm = temp_thetas.tail(q).transpose() * temp_thetas.tail(q) ;

temp_lambdas[1] = Rcpp :: as<double>(Rcpp :: rgamma(1, a[1] + q * 0.5 ,
                        1.0 / ( b[1]  + SecondFNorm * 0.5 ) ) )    ;

V_inv.topLeftCorner(p, p) =temp_lambdas[0] * tXX + Sigma_beta_inv ;
V_inv.topRightCorner(p, q) = temp_lambdas[0] * tXZ ;
V_inv.bottomLeftCorner(q, p) = temp_lambdas[0] * tZX ;
V_inv.bottomRightCorner(q, q) = temp_lambdas[0] * tZZ + temp_lambdas[1] * identity_q ;
V = V_inv.inverse() ;

zeta.head(p) = temp_lambdas[0] * tXy + Sigma_beta_inv * mu_beta ;
zeta.tail(q) = temp_lambdas[0] * tZy ;

normals = Rcpp::as<MapVecd>(Rcpp::rnorm(p+q)) ;

temp_thetas = V * zeta + V.llt().matrixL() * normals ;

}

thetas.row(i) = temp_thetas ;
sigmas.row(i) = 1 / temp_lambdas.array().sqrt() ;
}


MatrixXd  betas = thetas.leftCols(p);
MatrixXd  us = thetas.rightCols(q);

return Rcpp::List::create(
Rcpp::Named("beta") = betas,
Rcpp::Named("group") = us,
Rcpp::Named("sigma") = sigmas);
'

GibbsLMMcpp = cxxfunction(signature(iterations = "int", Burnin = "int", nthin = "int",
                                    Response = "numeric",
                                    ModelMatrixX = "numeric",
                                    ModelMatrixZ = "numeric",
                                    prior_mean_beta = "numeric",
                                    prior_cov_beta = "numeric",
                                    prior_gamma_shape = "numeric",
                                    prior_gamma_rate = "numeric",
                                    starttheta = "numeric"),
                          src_eigen_imp, plugin="RcppEigen")


##################
#### JAGS Code ###
##################


cat( "
     var
     Response[N], Beta[P], MIN[N], u[q], cutoff[q+1],
     prior.mean[P], prior.precision[P, P], mu[N],
     tau_prior_shape[2], tau_prior_rate[2], tau_e, tau_u, tau[N];
```

```
model{
# Likelihood specification
for(i in 1:q){
for(k in (cutoff[i]+1):cutoff[i+1]){
Response[k] ~ dnorm(mu[k], tau[k])
mu[k] <- Beta[1] + Beta[2] * MIN[k] + u[i]
tau[k] <- 1/((1 / tau_e) + (1 / tau_u))
}
u[i] ~ dnorm(0, tau_u)
}

# Prior specification
Beta[] ~ dmnorm(prior.mean[], prior.precision[,])

tau_u ~ dgamma(tau_prior_shape[1], tau_prior_rate[1])
tau_e ~ dgamma(tau_prior_shape[2], tau_prior_rate[2])
sigma_e <- sqrt(1 / tau_e)
sigma_u <- sqrt(1 / tau_u)
}",
file="LMM_nba.jags")
```

Listing C.4: Linear Mixed Model with Normally Distributed Random effects R Source Code

GibbsLMM input description:

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `ModelMatrixZ`: A sparse matrix filled with 0's and 1's that associates each observation to a random effect
- `prior_mean_beta`: A numeric vector that provides the mean parameter for the prior distribution of β
- `prior_cov_beta`: A numeric matrix that provides the covariance matrix for the prior distribution of β
- `prior_gamma_shape`: A numeric vector that provides the shape parameter for the prior distribution of τ_e and τ_u respectively
- `prior_gamma_rate`: A numeric vector that provides the rate parameter for the prior distribution of τ_e and τ_u respectively
- `start_theta`: A concatenated numeric vector of initial values for MCMC of β and u

GibbsLMMcpp input description:

- `iterations`: Net length of MCMC chain for main sample
- `Burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model

- `ModelMatrixZ`: A sparse matrix filled with 0's and 1's that associates each observation to a random effect
- `prior_mean_beta`: A numeric vector that provides the mean parameter for the prior distribution of β
- `prior_cov_beta`: A numeric matrix that provides the covariance matrix for the prior distribution of β
- `prior_gamma_shape`: A numeric vector that provides the shape parameter for the prior distribution of τ_e and τ_u respectively
- `prior_gamma_rate`: A numeric vector that provides the rate parameter for the prior distribution of τ_e and τ_u respectively
- `starttheta`: A concatenated numeric vector of initial values for MCMC of β and u

C.2.2 R/Rcpp/JAGS Workflow

```
getwd()
setwd()



#call libraires and source
# install.packages("nlme")
# install.packages("Rcpp")
# install.packages("RcppEigen")
# install.packages("coda")
# install.packages("inline")
# install.packages("rjags")


library(nlme)
library(Rcpp)
library(RcppEigen)
library(coda)
library(inline)
library(rjags)
library(matrixcalc)

source("LinearMixedModel_NBASource_2016-08-09.R") #calls the source f

getwd()
setwd()

nba <- read.csv(file="NBA2015Data.csv", header=TRUE)

# We won't use attach(nba) in this example

plot(nba$MIN, nba$PTS, xlab="minutes", ylab="points per game")



###################################
#      Frequentist analysis       #
###################################

# Let's model log(PTS) vs MIN; log = natural log
# But now we treat the TEAM variable as a random effect

# Be careful a few players have PTS=0
which(nba$PTS==0)

# Let's look at their data

nba[which(nba$PTS==0), ]

```

```r
#  Let's remove some problematic observations from the data set
nba.r=subset(nba, nba$PTS>0 & nba$GP>5)
nba.r <- nba.r[order(nba.r$TEAM), ]
# sort data by team ; this is very important for our "home-made" Gibbs sampler.

nba.r$log.PTS <- log(nba.r$PTS)

dim(nba)
dim(nba.r)

# Consider the following random intercept model
log.fit.mixed <- lme( log.PTS ~ MIN, random = ~1 | TEAM, data = nba.r)
summary(log.fit.mixed)
coefficients(log.fit.mixed) # beta_1 + u_j
log.fit.mixed$coeff$random # u_j
intervals(log.fit.mixed)

team.size <- as.numeric(table(nba.r$TEAM))
cutoff <- cumsum(c(0,team.size))


########################################
#  Bayesian Analysis Reference Prior  #
########################################
log.fit.mixed <- lme( log.PTS ~ MIN, random = ~1 | TEAM, data = nba.r)

# Make sure that the data are sorted by TEAM
ModelMatrixY <- log(nba.r$PTS)

log.fit.fixed <- lm(log(PTS) ~ MIN, data=nba.r)
ModelMatrixX <- model.matrix(log.fit.fixed) # trick to get the X matrix

log.fit.random <- lm(log(PTS) ~ TEAM - 1, data=nba.r)
ModelMatrixZ <- model.matrix(log.fit.random) #  trick to get Z matrix

beta.hat <- as.vector(log.fit.mixed$coeff$fixed)
u.hat <- coefficients(log.fit.mixed)[ , 1] - as.vector(log.fit.mixed$coeff$fixed)[1]
start.thetas <- c(beta.hat, u.hat)

prior.mean.beta = rep(0.0, ncol(ModelMatrixX))
prior.cov.beta = diag(ncol(ModelMatrixX)) * 100
tau.prior.rate = 1
tau.prior.shape = 1

beta.hat <- solve(t(ModelMatrixX)%*%ModelMatrixX) %*% t(ModelMatrixX) %*% ModelMatrixY

iterations = 5

### R Code ###
set.seed(999)
for(l in 1:iterations){
  start.time<-Sys.time()
 MCMC <- GibbsLMM(iterations = 500000, burnin = 500000, nthin = 1,
                  Response = ModelMatrixY, ModelMatrixX = ModelMatrixX,
                  ModelMatrixZ = ModelMatrixZ, prior_mean_beta = beta.hat,
                  prior_cov_beta = prior.cov.beta,
                  prior_gamma_shape = c(0.001, 0.001),
                  prior_gamma_rate = c(0.001, 0.001), start_theta = start.thetas)
  print(Sys.time() - start.time)

  print("###################################################")
  print("###################################################")
  print(paste("############ This is iteration: ", l,"##############"))
  print("###################################################")
  print("###################################################")

  for(r in names(MCMC)){
    print(summary(as.mcmc(MCMC[[r]])))
  }
  # write.csv(x = MCMC,
  #           file = paste("LinearMixedModelNBAData_",l,"_iterationR_2016-07-20.csv",
  #                        sep=""))
}
```

```

### Rcpp Code ###

set.seed(999)
for(l in 1 :iterations){
  start.time<-Sys.time()
 MCMC <- GibbsLMMcpp(iterations = 500000, Burnin = 500000, nthin = 1,
                       Response = ModelMatrixY,
                       ModelMatrixX = ModelMatrixX,
                       ModelMatrixZ = ModelMatrixZ,
                       prior_mean_beta = beta.hat,
                       prior_cov_beta = prior.cov.beta,
                       prior_gamma_shape = c(0.001,0.001),
                       prior_gamma_rate = c(0.001, 0.001),
                       starttheta = start.thetas)
  print(Sys.time() - start.time)

  print("##################################################")
  print("##################################################")
  print(paste("############ This is iteration: ", l,"##############"))
  print("##################################################")
  print("##################################################")

  for(r in names(MCMC)){
    print(summary(as.mcmc(MCMC[[r]])))
  }
  # write.csv(x = MCMC,
  #           file = paste("LinearMixedModelNBAData_",l,"_iterationRcpp_2016-07-20.csv",
  #                        sep=""))
}


### JAGS Code ###

set.seed(999)
for(l in 1 :iterations){
  jagsfit <- jags.model(file = "LMM_nba.jags",
                        data = list('Response' = ModelMatrixY,
                                    'MIN' = nba.r$MIN,
                                    'cutoff' = cutoff,
                                    'N' = length(ModelMatrixY),
                                    'P' = ncol(ModelMatrixX),
                                    'q' = ncol(ModelMatrixZ),
                                    'prior.mean' = as.vector(prior.mean.beta),
                                    'prior.precision' = solve(prior.cov.beta),
                                    'tau_prior_shape' = c(0.001, 0.001),
                                    'tau_prior_rate' = c(0.001, 0.001)),
                        inits = list('Beta'= as.vector(beta.hat),'tau_e' = 1, 'tau_u' = 1,
                                     'u' = u.hat),
                        n.chains=1,
                        n.adapt=0
  )
  start.time <- Sys.time()

  update(jagsfit, 500000) # Obtain first 100,000 (burnin draws)

 MCMC.out <- coda.samples(jagsfit,
                          var = c("Beta", "u", "sigma_e", "sigma_u"),
                          n.iter = 500000,  # Obtain the main 100,000 draws
                          thin = 1)
  print(Sys.time()  - start.time       )
  print(Sys.time()  - start.time)

  print("##################################################")
  print("##################################################")
  print(paste("############ This is iteration: ", l,"##############"))
  print("##################################################")
  print("##################################################")

  print(summary(MCMC.out)) # Notice the iterations being used
  # write.csv(x = MCMC,
  #           file = paste("LinearMixedModelNBAData_",l,"_iterationJAGS_2016-07-20.csv",
  #                        sep=""))
}
```

Listing C.5: Linear Mixed Model with Normally Distributed Random effects R Work Flow

jagsfit input description:

- `Response`: A numeric vector of observed data for linear model
- `MIN`: A numeric vector of observed data for the predictor variable MIN for linear model of NBA
- `cutoff`: A numeric vector of cumulatively summed entries of the number of players in each team of the NBA 2015 season data used for the random effect
- `N`: Sample size of the observed values
- `P`: The number of columns for the model matrix of linear model, i.e. (Number of predictors used for linear model) + 1
- `q`: The number of teams considered for the model based on the data set
- `prior.mean`: A numeric vector for the mean parameter of the normal distribution of β
- `prior.precision`: A numeric matrix for the covariance matrix parameter of the normal distribution of β
- `tau_prior_shape`: A numeric value that provides the shape parameter for the prior distribution of τ
- `tau_prior_rate`: A numeric value that provides the rate parameter for the prior distribution of τ
-
- `Beta`: A numeric vector of initial values for MCMC for β
- `tau_e`: A numeric value for initializing MCMC for τ_e
- `tau_u`: A numeric value for initializing MCMC for τ_u
- `u`: A numeric vector of initial values for MCMC of u

C.2.3 Julia

```
using Distributions, DataFrames
srand(1234)
function GibbsLMM(iterations, burnin, nthin, Response, ModelMatrixX, ModelMatrixZ,
    prior_mean_beta, prior_cov_beta, prior_gamma_shape, prior_gamma_rate, start_theta)
  X = ModelMatrixX
  Z = ModelMatrixZ
  y = Response
  Sigma_beta = prior_cov_beta
  Sigma_beta_inv = inv(Sigma_beta)
  mu_beta = prior_mean_beta
  a = prior_gamma_shape
  b = prior_gamma_rate
  W = [X Z]
  N = length(y)
  p = size(X, 2)
  q = size(Z, 2)

  tXX = X' * X
  tXZ = X' * Z
  tZX = Z' * X
  tZZ = Z' * Z
  tXy = X' * y
  tZy = Z' * y

  thetas = fill(0.0, iterations, p+q)
  lambdas = fill(0.0, iterations, 2)
  sigmas = fill(0.0, iterations, 2)
  temp_thetas = start_theta
  temp_lambdas = fill(0.0, 1, 2)
  temp_sigmas = fill(0.0, 1, 2)
  eta = fill(0.0, q)
  postrate_e = 0.0
  postshape_e = 0.0
  V_inv = fill(0.0, p+q, p+q)
  D_eta = diagm(fill(1.0,q))
  postshape_e = a[1] + N * 0.5
  postshape_u = a[2] + q * 0.5
  for i in 1:burnin
      postrate_e = b[1] + (vecnorm(y - W * temp_thetas)^2)/2
      postrate_u = b[2] + (vecnorm(D_eta * temp_thetas[(p+1):end])^2)/2

      temp_lambdas[1] = rand(Gamma(postshape_e, 1.0/postrate_e))
      temp_lambdas[2] = rand(Gamma(postshape_u, 1.0/postrate_u))
      temp_sigmas[1] = 1.0/sqrt(temp_lambdas[1])
      temp_sigmas[2] = 1.0/sqrt(temp_lambdas[2])

      topleft = temp_lambdas[1] * tXX + Sigma_beta_inv
      topright = temp_lambdas[1] * tXZ
      botleft = temp_lambdas[1] * tZX
      botright = temp_lambdas[1] * tZZ + temp_lambdas[2] * D_eta

      V_inv = [topleft topright; botleft botright]
      V = inv(V_inv)

      term1 = (temp_lambdas[1] * tXy) + (Sigma_beta_inv * mu_beta)
      term2 = temp_lambdas[1] * tZy
      zeta = [term1; term2]

      Vchol=transpose(chol(V))
      temp_thetas = (V * zeta) + (Vchol * rand(Normal(0,1),p+q))

  end

  for i in 1:iterations
    for nth in nthin
      postrate_e = b[1] + (vecnorm(y - W * temp_thetas)^2)/2
      postrate_u = b[2] + (vecnorm(D_eta^0.5 * temp_thetas[(p+1):end])^2)/2

      temp_lambdas[1] = rand(Gamma(postshape_e, 1.0/postrate_e))
      temp_lambdas[2] = rand(Gamma(postshape_u, 1.0/postrate_u))
      temp_sigmas[1] = 1.0/sqrt(temp_lambdas[1])
      temp_sigmas[2] = 1.0/sqrt(temp_lambdas[2])

      topleft = temp_lambdas[1] * tXX + Sigma_beta_inv
      topright = temp_lambdas[1] * tXZ
      botleft = temp_lambdas[1] * tZX
      botright = temp_lambdas[1] * tZZ + temp_lambdas[2] * D_eta

      V_inv = [topleft topright; botleft botright]
      V = inv(V_inv)

      term1 = (temp_lambdas[1] * tXy) + (Sigma_beta_inv * mu_beta)
      term2 = temp_lambdas[1] * tZy
      zeta = [term1; term2]

      Vchol=transpose(chol(V))
      temp_thetas = (V * zeta) + (Vchol * rand(Normal(0,1),p+q))
    end
      thetas[i,:] = temp_thetas'
      lambdas[i, :] = temp_lambdas
      sigmas[i, :] = temp_sigmas
  end

  return [thetas sigmas]
end

# Create .csv files from R and import into Julia
Y = readtable("nbadaty.csv") # Response variable
```

```
X = readtable("nbadatx.csv")
Z = readtable("nbadatz.csv")
initialtheta = readtable("nbadatinit.csv") # list of starting values for chain
DatX = convert(Array{Float64,2}, X)[:,2:end]
DatY = convert(Array{Float64,2}, Y)[:,2:end]
DatZ = convert(Array{Float64,2}, Z)[:,2:end]
thetastart = convert(Array{Float64,2}, initialtheta)[:,2]



iterations = 5
for l in 1:iterations
  @time dataoutput = GibbsLMM(500000, 500000, 1, DatY, DatX, DatZ, [0.26872485; 0.07814486],
    eye(2) * 100, [0.001, 0.001], [0.001,0.001], thetastart)
  describe(convert(DataFrame, dataoutput))
  # writedlm(string("LME_normal-effects_",l,".txt"),dataoutput)
end
```

Listing C.6: Julia Code

GibbsLMM input description:

- `iterations`: Net length of MCMC chain for main sample
- `Burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `ModelMatrixZ`: A sparse matrix filled with 0's and 1's that associates each observation to a random effect
- `prior_mean_beta`: A numeric vector for the mean parameter of the normal distribution of β
- `prior_cov_beta`: A numeric matrix for the covariance matrix parameter of the normal distribution of β
- `prior_gamma_shape`: A numeric vector that provides the shape parameter for the prior distribution of τ_e and τ_u respectively
- `prior_gamma_rate`: A numeric vector that provides the rate parameter for the prior distribution of τ_e and τ_u respectively
- `start_theta`: A concatenated numeric vector of initial values for MCMC of β and u

C.2.4 MATLAB

```
function [thetas, sigmas] = GibbsLMM(iterations, burnin, nthin, Response, ModelMatrixX,
    ModelMatrixZ, prior_mean_beta, prior_cov_beta, prior_gamma_shape, prior_gamma_rate,
    start_theta)

X = ModelMatrixX;
Z = ModelMatrixZ;
y = Response;
Sigma_beta = prior_cov_beta;
Sigma_beta_inv = inv(Sigma_beta);
mu_beta = prior_mean_beta;
a = prior_gamma_shape;
b = prior_gamma_rate;
W = [X, Z];
N = length(y);
p = size(X, 2);
q = size(Z, 2);

```

```
tXX = X' * X;
tXZ = X' * Z;
tZX = Z' * X;
tZZ = Z' * Z;
tXy = X' * y;
tZy = Z' * y;

thetas = repmat(0.0, iterations, p+q);
lambdas = repmat(0.0, iterations, 2);
sigmas = repmat(0.0, iterations, 2);
temp_thetas = start_theta;
temp_lambdas = repmat(0.0,1,2);
temp_sigmas = repmat(0.0,1,2);
eta = repmat(0.0, q,1);
postrate_e = 0.0 ;
postshape_e = 0.0;

V_inv = repmat(0.0, p+q,p+q);
D_eta = eye(q);
postshape_e = a(1) + N * 0.5;
postshape_u = a(2) + q * 0.5;
for i = 1:burnin
    postrate_e = b(1) + (norm(y - W * temp_thetas)^2)/2;
    postrate_u = b(2) + (norm(D_eta^0.5 * temp_thetas((p+1):end))^2)/2;

    temp_lambdas(1) =gamrnd(postshape_e, 1/postrate_e,1);
    temp_lambdas(2) = gamrnd(postshape_u, 1.0/postrate_u);
    temp_sigmas(1) = 1.0/sqrt(temp_lambdas(1));
    temp_sigmas(2) = 1.0/sqrt(temp_lambdas(2));

    topleft = temp_lambdas(1) * tXX + Sigma_beta_inv;
    topright = temp_lambdas(1) * tXZ;
    botleft = temp_lambdas(1) * tZX;
    botright = temp_lambdas(1) * tZZ + temp_lambdas(2) * D_eta;

    V_inv = [topleft, topright; botleft, botright];
   V = inv(V_inv);

    term1 = (temp_lambdas(1) * tXy) + (Sigma_beta_inv * mu_beta);
    term2 = temp_lambdas(1) * tZy;
    zeta = [term1; term2];

    Vchol=transpose(chol(V));
    temp_thetas = (V * zeta) + (Vchol * normrnd(0,1,p+q,1) );
end
for i = 1:iterations
    for nth = 1:nthin
        postrate_e = b(1) + (norm(y - W * temp_thetas)^2)/2;
        postrate_u = b(2) + (norm(D_eta^0.5 * temp_thetas((p+1):end))^2)/2;

        temp_lambdas(1) =gamrnd(postshape_e, 1/postrate_e,1);
        temp_lambdas(2) = gamrnd(postshape_u, 1.0/postrate_u);
        temp_sigmas(1) = 1.0/sqrt(temp_lambdas(1));
        temp_sigmas(2) = 1.0/sqrt(temp_lambdas(2));

        topleft = temp_lambdas(1) * tXX + Sigma_beta_inv;
        topright = temp_lambdas(1) * tXZ;
        botleft = temp_lambdas(1) * tZX;
        botright = temp_lambdas(1) * tZZ + temp_lambdas(2) * D_eta;

        V_inv = [topleft, topright; botleft, botright];
       V = inv(V_inv);

        term1 = (temp_lambdas(1) * tXy) + (Sigma_beta_inv * mu_beta);
        term2 = temp_lambdas(1) * tZy;
        zeta = [term1; term2];

        Vchol=transpose(chol(V));
        temp_thetas = (V * zeta) + (Vchol * normrnd(0,1,p+q,1) );
    end
    thetas(i,:) = temp_thetas';
    lambdas(i, :) = temp_lambdas;
    sigmas(i, :) = temp_sigmas;
```

```
end

end
```

Listing C.7: MATLAB code

GibbsLMM input description:

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `ModelMatrixZ`: A sparse matrix filled with 0's and 1's that associates each observation to a random effect
- `prior_mean_beta`: A numeric vector for the mean parameter of the normal distribution of β
- `prior_cov_beta`: A numeric matrix for the covariance matrix parameter of the normal distribution of β
- `prior_gamma_shape`: A numeric vector that provides the shape parameter for the prior distribution of τ_e and τ_u respectively
- `prior_gamma_rate`: A numeric vector that provides the rate parameter for the prior distribution of τ_e and τ_u respectively
- `start_theta`: A concatenated numeric vector of initial values for MCMC of β and u

C.3 Proper Priors – t-Distributed Random Effects

C.3.1 Source code

```

##################
### R Function ###
##################

GibbsLMMt = function(iterations, burnin, nthin, Response, ModelMatrixX, ModelMatrixZ,
                     prior_mean_beta, prior_cov_beta, prior_gamma_shape,
                     prior_gamma_rate, df, start_theta){

  X <- ModelMatrixX
  y <- Response
  Z <- ModelMatrixZ
  Sigma_beta <- prior_cov_beta
  Sigma_beta_inv <- solve(prior_cov_beta)
  mu_beta <- prior_mean_beta
  a <- prior_gamma_shape # Shape for e and u
  b <- prior_gamma_rate # Rate for e and u


  N <- length(y) # sample size
  p <- ncol(X) # number of columns of X
  q <- ncol(Z) # number of columns of Z
  W <- cbind(X, Z)

  tXX <- t(X) %*% X
  tXZ <- t(X) %*% Z
  tZX <- t(Z) %*% X
  tZZ <- t(Z) %*% Z
```

```
 tXy <- t(X) %*% y
 tZy <- t(Z) %*% y

 thetas <- matrix(NA, nrow = iterations, ncol = {p+q})
 lambdas <- matrix(NA, nrow = iterations, ncol = 2)
 temp_thetas = start_theta # (beta, u)
 temp_lambdas = c(0,0) #(lambda_e, lambda_u)
 eta = rep(0, q)
 V_inv = matrix(NA , nrow = p + q, ncol = p + q)
 D_eta = diag(q)
 for (j in 1 : burnin){
   test = y- (W %*% temp_thetas)
   Fnorm = norm(x = test, type="F")
   temp_lambdas[1] = rgamma(1, a[1] + N * 0.5, b[1] + (Fnorm^2) * 0.5)
   SecondFnorm = norm(x = D_eta %*% temp_thetas[(p+1):(p+q)], type = "F")^2
   temp_lambdas[2] = rgamma(1, a[2] + q * 0.5, b[2] + SecondFnorm*0.5)
   for(l in 1:q){
     eta[l] = sqrt(rgamma(1, (df+1)* 0.5, (df+temp_lambdas[2]*temp_thetas[p+l]^2)*0.5))
   }

   T_lambda = temp_lambdas[1] * tXX + Sigma_beta_inv
   T_lambda_inv = chol2inv(chol(T_lambda))

   M_lambda = diag(N) - temp_lambdas[1] * X %*% T_lambda_inv %*% t(X)

   D_eta = diag(eta)

   Q_lambda_eta = temp_lambdas[1] * t(Z) %*% M_lambda %*% Z + temp_lambdas[2] * D_eta

   V_inv[1:p, 1:p] <- T_lambda
   V_inv[1:p, {p+1}:{p+q}] <-  temp_lambdas[1] * tXZ
   V_inv[{p+1}:{p+q}, 1:p] <-  temp_lambdas[1] * tZX
   V_inv[{p+1}:{p+q}, {p+1}:{p+q}] <-  temp_lambdas[1] * tZZ + temp_lambdas[2] * D_eta

   V <- chol2inv(chol(V_inv))

   NextTerm1 <- temp_lambdas[1] * tXy + Sigma_beta_inv %*% mu_beta
   NextTerm2 <- temp_lambdas[1] * tZy

   zeta <- c(NextTerm1, NextTerm2)

   Vchol <- t(chol(V)) # cholesky decomposition
   temp_thetas <- V %*% zeta + Vchol %*% rnorm(p+q)

 }

 for(i in 1 : iterations){
   for (j in 1 : nthin){
     test= y- (W %*% temp_thetas)
     Fnorm = norm(x = test, type="F")
     temp_lambdas[1] = rgamma(1, a[1] + N * 0.5, b[1] + (Fnorm^2) * 0.5)
     SecondFnorm = norm(x = D_eta %*% temp_thetas[(p+1):(p+q)], type = "F")^2
     temp_lambdas[2] = rgamma(1, a[2] + q * 0.5, b[2] + SecondFnorm*0.5)
     for(l in 1:q){
       eta[l] = sqrt(rgamma(1, (df+1)* 0.5, (df+temp_lambdas[2]*temp_thetas[p+l]^2)*0.5))
     }

     T_lambda = temp_lambdas[1] * tXX + Sigma_beta_inv
     T_lambda_inv =  chol2inv(chol(T_lambda))

     M_lambda = diag(N) - temp_lambdas[1] * X %*% T_lambda_inv %*% t(X)

     D_eta = diag(eta)

     Q_lambda_eta = temp_lambdas[1] * t(Z) %*% M_lambda %*% Z + temp_lambdas[2] * D_eta

     V_inv[1:p, 1:p] <- T_lambda
     V_inv[1:p, {p+1}:{p+q}] <-  temp_lambdas[1] * tXZ
     V_inv[{p+1}:{p+q}, 1:p] <-  temp_lambdas[1] * tZX
     V_inv[{p+1}:{p+q}, {p+1}:{p+q}] <-  temp_lambdas[1] * tZZ + temp_lambdas[2] * D_eta

     V <-  chol2inv(chol(V_inv))

```

```
      NextTerm1 <- temp_lambdas[1] * tXy + Sigma_beta_inv %*% mu_beta
      NextTerm2 <- temp_lambdas[1] * tZy

      zeta <- c(NextTerm1, NextTerm2)

      Vchol <- t(chol(V)) # cholesky decomposition
      temp_thetas <- V %*% zeta + Vchol %*% rnorm(p+q)
    }
    thetas[i , ] <- temp_thetas
    lambdas[i , ] <- temp_lambdas
  }

  sigmas <- 1 / sqrt(lambdas)

  return( list( beta = thetas[, 1:p], group = thetas[, {p+1}: {p+q}], sigma = sigmas) )
}



#####################
### Rcpp Function ###
#####################

src_eigen_imp<- '

using Eigen :: Map ;
using Eigen :: MatrixXd ;
using Eigen :: VectorXd ;
using Eigen :: Vector2d ;
using Rcpp :: as ;

typedef Eigen :: Map<Eigen::MatrixXd> MapMatd ;
typedef Eigen :: Map<Eigen::VectorXd> MapVecd ;

int MCMCiter = Rcpp::as<int>(iterations);
int burnin = Rcpp :: as<int>(Burnin);
int n_thin = Rcpp :: as<int>(nthin);
int df = Rcpp :: as<int>(DF);

Rcpp :: NumericMatrix Xc(ModelMatrixX) ;
Rcpp :: NumericMatrix Zc(ModelMatrixZ) ;
Rcpp :: NumericMatrix Sigma_betac(prior_cov_beta) ;
Rcpp :: NumericVector yc(Response) ;
Rcpp :: NumericVector mu_betac(prior_mean_beta) ;

const MapMatd X(Rcpp :: as<MapMatd>(Xc)) ;
const MapMatd Z(Rcpp :: as<MapMatd>(Zc)) ;
const MapMatd Sigma_beta(Rcpp :: as<MapMatd>(Sigma_betac)) ;
const MapVecd y(Rcpp :: as<MapVecd>(yc)) ;
const MapVecd mu_beta(Rcpp :: as<MapVecd>(mu_betac)) ;

const MatrixXd Sigma_beta_inv = Sigma_beta.inverse() ;

int N = y.rows(), p = X.cols(), q = Z.cols() ;

Rcpp :: NumericVector starttheta c(starttheta) ;
const MapVecd     start_theta(Rcpp::as<MapVecd>(startthetac)) ;

Rcpp :: NumericVector ac(prior_gamma_shape) ;
Rcpp :: NumericVector bc(prior_gamma_rate) ;

Rcpp :: NumericMatrix D_Eta(q,q) ;

const MapVecd a(Rcpp::as<MapVecd>(ac)) ;
const MapVecd b(Rcpp::as<MapVecd>(bc)) ;

const MatrixXd tXX = X.transpose() * X ;
const MatrixXd tXZ = X.transpose() * Z ;
const MatrixXd tZX = Z.transpose() * X ;
const MatrixXd tZZ = Z.transpose() * Z ;
const VectorXd tXy = X.transpose() * y ;
const VectorXd tZy = Z.transpose() * y ;

```

```
MatrixXd thetas(MCMCiter, p+q) ;
MatrixXd sigmas(MCMCiter, 2) ;
thetas.col(0) = start_theta ;
VectorXd temp_thetas = start_theta ;
VectorXd temp_lambdas(2); temp_lambdas << 0, 0 ;

VectorXd zeta(p+q) ;
MatrixXd T_lambda(p,p) ;
MatrixXd T_lambda_inv(p,p);
MatrixXd M_lambda(N,N) ;
MatrixXd Q_lambda_eta(q,q) ;
MatrixXd V_inv(p+q, p+q) ;
MatrixXd V(p+q, p+q) ;
const MatrixXd identity_N = MatrixXd :: Identity(N,N) ;
MatrixXd test(N,1) ;

MatrixXd W(N, p+q) ;
W.leftCols(p) = X ;
W.rightCols(q) = Z ;

double FNorm ;
double SecondFNorm ;
MatrixXd D_eta = Rcpp :: as<MapMatd>(D_Eta);

RNGScope scp ;
Rcpp :: Function rnorm("rnorm") ;
Rcpp :: Function rgamma("rgamma") ;
Rcpp :: Function fnorm("frobenius.norm") ;

MapVecd normals = Rcpp::as<MapVecd>(Rcpp::rnorm(p+q)) ;

for(int j = 0; j < burnin; j++){

test = y - (W * temp_thetas) ;

FNorm = Rcpp :: as<double>(fnorm(test)) ;

temp_lambdas[0] = Rcpp :: as<double>(Rcpp :: rgamma(1, a[0] + N * 0.5 ,
                                  1.0 / (b[0] + pow(FNorm , 2.0 ) * 0.5 ) ) ) ;

SecondFNorm = temp_thetas.tail(q).transpose() * D_eta * temp_thetas.tail(q) ;

temp_lambdas[1] = Rcpp :: as<double>(Rcpp :: rgamma(1, a[1] + q * 0.5 ,
                                  1.0 / ( b[1]  + SecondFNorm * 0.5 ) ) )   ;

T_lambda = temp_lambdas[0] * tXX + Sigma_beta_inv ;

T_lambda_inv = T_lambda.inverse() ;

M_lambda = identity_N - temp_lambdas[0] * X * T_lambda_inv * X.transpose() ;

for (int k = 0; k < q; k++) {
D_Eta(k,k) = sqrt(Rcpp :: as<double>(Rcpp :: rgamma(1 , (df + 1) * 0.5 ,
                    1.0 / ((df + temp_lambdas[1] * pow(temp_thetas[p + k], 2.0 ) ) * 0.5 )))) ;
}

D_eta = Rcpp :: as<MapMatd>(D_Eta) ;
Q_lambda_eta = temp_lambdas[0] * Z.transpose() * M_lambda * Z + temp_lambdas[1] * D_eta ;

V_inv.topLeftCorner(p, p) = T_lambda  ;
V_inv.topRightCorner(p, q) = temp_lambdas[0] * tXZ ;
V_inv.bottomLeftCorner(q, p) = temp_lambdas[0] * tZX ;
V_inv.bottomRightCorner(q, q) = temp_lambdas[0] * tZZ + temp_lambdas[1] * D_eta ;
V = V_inv.inverse() ;

zeta.head(p) = temp_lambdas[0] * tXy + Sigma_beta_inv * mu_beta ;
zeta.tail(q) = temp_lambdas[0] * tZy ;

normals = Rcpp::as<MapVecd>(Rcpp::rnorm(p+q)) ;

temp_thetas = V * zeta + V.llt().matrixL() * normals ;

}
```

```
for(int i = 0; i < MCMCiter; i++){
for(int j = 0; j < n_thin; j++){

test = y - (W * temp_thetas) ;

FNorm = Rcpp :: as<double>(fnorm(test)) ;

temp_lambdas[0] = Rcpp :: as<double>(Rcpp :: rgamma(1, a[0] + N * 0.5 ,
                                    1.0 / (b[0] + pow(FNorm , 2.0 ) * 0.5 ) ) ) ;

SecondFNorm = temp_thetas.tail(q).transpose() * D_eta * temp_thetas.tail(q) ;

temp_lambdas[1] = Rcpp :: as<double>(Rcpp :: rgamma(1, a[1] + q * 0.5 ,
                                  1.0 / ( b[1]  + SecondFNorm * 0.5 ) ) )    ;

T_lambda = temp_lambdas[0] * tXX + Sigma_beta_inv ;

T_lambda_inv = T_lambda.inverse() ;

M_lambda = identity_N - temp_lambdas[0] * X * T_lambda_inv * X.transpose() ;

for (int k = 0; k < q; k++) {
D_Eta(k,k) = sqrt(Rcpp :: as<double>(Rcpp :: rgamma(1 , (df + 1) * 0.5 ,
                 1.0 / ((df + temp_lambdas[1] * pow(temp_thetas[p + k], 2.0 ) ) * 0.5 )))) ;
}

D_eta = Rcpp :: as<MapMatd>(D_Eta) ;

Q_lambda_eta = temp_lambdas[1] * Z.transpose() * M_lambda * Z + temp_lambdas[1] * D_eta ;

V_inv.topLeftCorner(p, p) = T_lambda  ;
V_inv.topRightCorner(p, q) = temp_lambdas[0] * tXZ ;
V_inv.bottomLeftCorner(q, p) = temp_lambdas[0] * tZX ;
V_inv.bottomRightCorner(q, q) = temp_lambdas[0] * tZZ + temp_lambdas[1] * D_eta ;

V = V_inv.inverse() ;

zeta.head(p) = temp_lambdas[0] * tXy + Sigma_beta_inv * mu_beta ;
zeta.tail(q) = temp_lambdas[0] * tZy ;

normals = Rcpp::as<MapVecd>(Rcpp::rnorm(p+q)) ;

temp_thetas = V * zeta + V.llt().matrixL() * normals ;
}

thetas.row(i) = temp_thetas ;
sigmas.row(i) = 1 / temp_lambdas.array().sqrt() ;
}


MatrixXd  betas = thetas.leftCols(p);
MatrixXd  us = thetas.rightCols(q);

return Rcpp::List::create(
Rcpp::Named("beta") = betas,
Rcpp::Named("group") = us,
Rcpp::Named("sigma") = sigmas);
'

GibbsLMMtcpp = cxxfunction(signature(iterations = "int", Burnin = "int", nthin = "int",
                                     Response = "numeric",
                                     ModelMatrixX = "numeric",
                                     ModelMatrixZ = "numeric",
                                     prior_mean_beta = "numeric",
                                     prior_cov_beta = "numeric",
                                     prior_gamma_shape = "numeric",
                                     prior_gamma_rate = "numeric", DF = "int",
                                     starttheta = "numeric"),
                           src_eigen_imp, plugin="RcppEigen")

###################
```

```

cat( "
     var
     Response[N], Beta[P], MIN[N], u[q], cutoff[q+1],
     prior.mean[P], prior.precision[P, P], mu[N],
     tau_prior_shape[2], tau_prior_rate[2], tau_e, tau_u, tau[N], df;
     model{
     # Likelihood specification
     for(i in 1:q){
     for(k in (cutoff[i]+1):cutoff[i+1]){
     Response[k] ~ dnorm(mu[k], tau[k])
     mu[k] <- Beta[1] + Beta[2] * MIN[k] + u[i]
     tau[k] <- 1/((1 / tau_e) + (1 / tau_u))
     }
     u[i] ~ dt(0, tau_u, df)
     }

     # Prior specification
     Beta[] ~ dmnorm(prior.mean[], prior.precision[,])

     tau_u ~ dgamma(tau_prior_shape[1], tau_prior_rate[1])
     tau_e ~ dgamma(tau_prior_shape[2], tau_prior_rate[2])
     sigma_e <- sqrt(1 / tau_e)
     sigma_u <- sqrt(1 / tau_u)
     }",
     file="LMM_nba.jags")
```

Listing C.8: Linear Mixed Model with t-Distributed Random effects R Source Code

GibbsLMMt input description:

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `ModelMatrixZ`: A sparse matrix filled with 0's and 1's that associates each observation to a random effect
- `prior_mean_beta`: A numeric vector that provides the mean parameter for the prior distribution of β
- `prior_cov_beta`: A numeric matrix that provides the covariance matrix for the prior distribution of β
- `prior_gamma_shape`: A numeric vector that provides the shape parameter for the prior distribution of τ_e and τ_u respectively
- `prior_gamma_rate`: A numeric vector that provides the rate parameter for the prior distribution of τ_e and τ_u respectively
- `df`: A numeric value for the degrees of freedom for the distribution of u
- `start_theta`: A concatenated numeric vector of initial values for MCMC of β and u

GibbsLMMtcpp input description:

- `iterations`: Net length of MCMC chain for main sample
- `Burnin`: Number of draws for MCMC chain to initialize before main sample

- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `ModelMatrixZ`: A sparse matrix filled with 0's and 1's that associates each observation to a random effect
- `prior_mean_beta`: A numeric vector that provides the mean parameter for the prior distribution of β
- `prior_cov_beta`: A numeric matrix that provides the covariance matrix for the prior distribution of β
- `prior_gamma_shape`: A numeric vector that provides the shape parameter for the prior distribution of τ_e and τ_u respectively
- `prior_gamma_rate`: A numeric vector that provides the rate parameter for the prior distribution of τ_e and τ_u respectively
- `DF`: A numeric value for the degrees of freedom for the distribution of u
- `start_theta`: A concatenated numeric vector of initial values for MCMC of β and u

C.3.2 R/Rcpp/JAGS Workflow

```r
getwd()
setwd()


#call libraires and source
# install.packages("nlme")
# install.packages("Rcpp")
# install.packages("RcppEigen")
# install.packages("coda")
# install.packages("inline")
# install.packaegs("matrixcalc")
# install.packages("rjags")


library(nlme)
library(Rcpp)
library(RcppEigen)
library(coda)
library(inline)
library(matrixcalc)
library(rjags)


source("LinearMixedModel_t-effects_NBAsource_2016-08-09.R")

getwd()
setwd()

nba <- read.csv(file="NBA2015Data.csv", header=TRUE)


####################################
#      Frequentist analysis       #
####################################

# Let's model log(PTS) vs MIN; log = natural log
# But now we treat the TEAM variable as a random effect

# Be careful a few players have PTS=0
which(nba$PTS==0)

```

```r
# Let's look at their data

nba[which(nba$PTS==0), ]

nba.r <- subset(nba, GP>5 & PTS>0)
nba.r <- nba.r[order(nba.r$TEAM), ]
# sort data by team ; this is very important for our "home-made" Gibbs sampler.
nba.r$log.PTS <- log(nba.r$PTS)
team.size <- as.numeric(table(nba.r$TEAM))
cutoff <- cumsum(c(0,team.size))


log.fit.mixed <- lme(log.PTS ~ MIN, random = ~1 | TEAM, data = nba.r)

# Make sure that the data are sorted by TEAM
ModelMatrixY <- log(nba.r$PTS)

log.fit.fixed <- lm(log(PTS) ~ MIN, data=nba.r)
ModelMatrixX <- model.matrix(log.fit.fixed) # trick to get the ModelMatrixX matrix

log.fit.random <- lm(log(PTS) ~ TEAM - 1, data=nba.r)
ModelMatrixZ <- model.matrix(log.fit.random) #  trick to get ModelMatrixZ matrix

beta.hat <- as.vector(log.fit.mixed$coeff$fixed)
u.hat <- coefficients(log.fit.mixed)[ , 1] - as.vector(log.fit.mixed$coeff$fixed)[1]
start.thetas <- c(beta.hat, u.hat)

prior.mean.beta = rep(0.0, ncol(ModelMatrixX))
prior.cov.beta = diag(ncol(ModelMatrixX)) * 100
tau.prior.rate = 1
tau.prior.shape = 1

beta.hat <- solve(t(ModelMatrixX)%*%ModelMatrixX) %*% t(ModelMatrixX) %*% ModelMatrixY



iterations = 1

### R Code ###
set.seed(999)

for(l in 1 :iterations){
  start.time<-Sys.time()
  output = GibbsLMMt(iterations = 500000, burnin = 500000, nthin = 1,
                     ModelMatrixX = ModelMatrixX, ModelMatrixZ = ModelMatrixZ,
                     Response = ModelMatrixY, prior_mean_beta = beta.hat,
                     prior_cov_beta = diag(ncol(ModelMatrixX))*100,
                     prior_gamma_shape = c(0.001, 0.001), prior_gamma_rate = c(0.001,0.001),
                     df = 100, start_theta = start.thetas)
  Sys.time() - start.time

  print("##################################################")
  print("##################################################")
  print(paste("############ This is iteration: ", l,"##############"))
  print("##################################################")
  print("##################################################")

  for(r in names(output)){
    print(summary(as.mcmc(output[[r]])))
  }
  write.csv(x = MCMC,
            file = paste("LME_T-Effects_NBAData_",l,"_iterationR_2016-07-29.csv",
                         sep=""))
}

### Rcpp Code ###

set.seed(999)

for(l in 1 :iterations){
  start.time<-Sys.time()
  output = GibbsLMMtcpp(iterations = 500000,  nthin = 1, Burnin = 500000, DF = 100,
                        starttheta = start.thetas, ModelMatrixX = ModelMatrixX,
                              ModelMatrixZ = ModelMatrixZ, Response = ModelMatrixY,
                              prior_mean_beta = beta.hat,
                              prior_cov_beta = diag(ncol(ModelMatrixX)) * 100,
                              prior_gamma_rate =  c(0.001, 0.001),
                              prior_gamma_shape = c(0.001,0.001) )
  Sys.time() - start.time

  print("#################################################")
  print("#################################################")
  print(paste("########### This is iteration: ", l,"#############"))
  print("#################################################")
  print("#################################################")

  for(r in names(output)){
    print(summary(as.mcmc(output[[r]])))
  }
  # write.csv(x = MCMC,
  #           file = paste("LME_T-Effects_NBAData_",l,"_iteration_Rcpp_2016-07-29.csv",
  #                        sep=""))
}

### JAGS Code ###

set.seed(999)
for(l in 1 :iterations){
  set.seed(999)
  jagsfit <- jags.model(file = "LMM_nba.jags",
                        data = list('Response' = ModelMatrixY,
                                    'MIN' = nba.r$MIN,
                                    'cutoff' = cutoff,
                                    'N' = length(ModelMatrixY),
                                    'P' = ncol(ModelMatrixX),
                                    'q' = ncol(ModelMatrixZ),
                                    'prior.mean' = as.vector(prior.mean.beta),
                                    'prior.precision' = solve(prior.cov.beta),
                                    'df' = 100,
                                    'tau_prior_shape' = c(0.001, 0.001),
                                    'tau_prior_rate' = c(0.001, 0.001)),
                        inits = list('Beta'= as.vector(beta.hat),'tau_e' = 1, 'tau_u' = 1,
                                     'u' = u.hat),
                        n.chains=1,
                        n.adapt=0
  )



  start.time <- Sys.time()
  update(jagsfit, 500000) # Obtain first 100,000 (burnin draws)

 MCMC.out <- coda.samples(jagsfit,
                           var = c("Beta", "u", "sigma_e", "sigma_u"),
                           n.iter = 500000,  # Obtain the main 100,000 draws
                           thin = 1)
  print(Sys.time() - start.time)

  # write.csv(x = as.mcmc(MCMC.out),
  #           file = paste("LinearMixedModelNBAData_multiple_length_",l
  #                        ,"_iterationJAGS_2016-08-03.csv",sep=""))

  print("#################################################")
  print("#################################################")
  print(paste("########### This is iteration: ", l,"#############"))
  print("#################################################")
  print("#################################################")

  print(summary(MCMC.out)) # Notice the iterations being used
}
```

Listing C.9: Linear Mixed Model with t-Distributed Random effects R Work Flow

jagsfit input description:

- `Response`: A numeric vector of observed data for linear model
- `MIN`: A numeric vector of observed data for the variable MIN for linear model of NBA
- `cutoff`: A numeric vector of cumulatively summed entries of the number of players in each team of the NBA 2015 season data used for the random effect
- `N`: Sample size of the observed values
- `P`: The number of columns for the model matrix of linear model, i.e. (Number of predictors used for linear model) + 1
- `q`: The number of teams considered for the model based on the data set
- `prior.mean`: A numeric vector for the mean parameter of the normal distribution of β
- `prior.precision`: A numeric matrix for the covariance matrix parameter of the normal distribution of β
- `tau_prior_shape`: A numeric value that provides the shape parameter for the prior distribution of τ
- `tau_prior_rate`: A numeric value that provides the rate parameter for the prior distribution of τ
- `df`: A numeric value for the degrees of freedom for the distribution of u
- `Beta`: A numeric vector of initial values for MCMC for β
- `tau_e`: A numeric value for initializing MCMC for τ_e
- `tau_u`: A numeric value for initializing MCMC for τ_u
- `u`: A numeric vector of initial values for MCMC of u

C.3.3 Julia

```julia
using Distributions, DataFrames
srand(1234)
function GibbsLMMT(iterations, burnin, nthin, Response, ModelMatrixX, ModelMatrixZ,
    prior_mean_beta, prior_cov_beta, prior_gamma_shape, prior_gamma_rate, df, start_theta)
  X = ModelMatrixX
  Z = ModelMatrixZ
  y = Response
  Sigma_beta = prior_cov_beta
  Sigma_beta_inv = inv(Sigma_beta)
  mu_beta = prior_mean_beta
  a = prior_gamma_shape
  b = prior_gamma_rate
  W = [X Z]
  N = length(y)
  p = size(X, 2)
  q = size(Z, 2)

  tXX = X' * X
  tXZ = X' * Z
  tZX = Z' * X
  tZZ = Z' * Z
  tXy = X' * y
  tZy = Z' * y

  iter = iterations
  thetas = fill(0.0, iter, p+q)
  lambdas = fill(0.0, iter, 2)
  sigmas = fill(0.0, iter, 2)
  temp_thetas = start_theta
  temp_lambdas = fill(0.0, 1, 2)
  temp_sigmas = fill(0.0, 1, 2)
  eta = fill(0.0, q)
  postrate_e = 0.0
  postshape_e = 0.0
  V_inv = fill(0.0, p+q, p+q)
  D_eta = diagm(fill(1.0,q))
  for i in 1:burnin
      postrate_e = b[1] + (vecnorm(y - W * temp_thetas)^2)/2
      postshape_e = a[1] + N * 0.5
      postrate_u = b[2] + (vecnorm(D_eta^0.5 * temp_thetas[(p+1):end])^2)/2
      postshape_u = a[2] + q * 0.5

      temp_lambdas[1] = rand(Gamma(postshape_e, 1.0/postrate_e))
      temp_lambdas[2] = rand(Gamma(postshape_u, 1.0/postrate_u))
      temp_sigmas[1] = 1.0/sqrt(temp_lambdas[1])
      temp_sigmas[2] = 1.0/sqrt(temp_lambdas[2])

      for(l in 1:q)
        etarate = (df + temp_lambdas[2] * (temp_thetas[p+l])^2 )/2
        eta[l] = rand(Gamma((df + 1.0) / 2, 1.0/etarate) )
      end

      T_lambda = temp_lambdas[1] * tXX + Sigma_beta_inv
      T_lambda_inv = inv(T_lambda)
      M_lambda = eye(N) - (temp_lambdas[1] * X * T_lambda_inv * X')
      D_eta = diagm(eta)
      Q_lambda_eta = (temp_lambdas[1] * Z' * M_lambda * Z) + temp_lambdas[2] * D_eta

      topleft = T_lambda
      topright = temp_lambdas[1] * tXZ
      botleft = temp_lambdas[1] * tZX
      botright = temp_lambdas[1] * tZZ + temp_lambdas[2] * D_eta

      V_inv = [topleft topright; botleft botright]
      V = inv(V_inv)

      term1 = (temp_lambdas[1] * tXy) + (Sigma_beta_inv * mu_beta)
      term2 = temp_lambdas[1] * tZy
      zeta = [term1; term2]

      Vchol=transpose(chol(V))
      temp_thetas = (V * zeta) + (Vchol * rand(Normal(0,1),p+q))

  end

  for i in 1:iter
    for nth in nthin
      postrate_e = b[1] + (vecnorm(y - W * temp_thetas)^2)/2
      postshape_e = a[1] + N * 0.5
      postrate_u = b[2] + (vecnorm(D_eta^0.5 * temp_thetas[(p+1):end])^2)/2
      postshape_u = a[2] + q * 0.5

      temp_lambdas[1] = rand(Gamma(postshape_e, 1.0/postrate_e))
      temp_lambdas[2] = rand(Gamma(postshape_u, 1.0/postrate_u))
      temp_sigmas[1] = 1.0/sqrt(temp_lambdas[1])
      temp_sigmas[2] = 1.0/sqrt(temp_lambdas[2])

      for(l in 1:q)
        etarate = (df + temp_lambdas[2] * (temp_thetas[p+l])^2 )/2
        eta[l] = rand(Gamma((df + 1.0) / 2, 1.0/etarate) )
      end

      T_lambda = temp_lambdas[1] * tXX + Sigma_beta_inv
      T_lambda_inv = inv(T_lambda)
      M_lambda = eye(N) - (temp_lambdas[1] * X * T_lambda_inv * X')
      D_eta = diagm(eta)
      Q_lambda_eta = (temp_lambdas[1] * Z' * M_lambda * Z) + temp_lambdas[2] * D_eta

      topleft = T_lambda
      topright = temp_lambdas[1] * tXZ
      botleft = temp_lambdas[1] * tZX
      botright = temp_lambdas[1] * tZZ + temp_lambdas[2] * D_eta
```

```julia

      V_inv = [topleft topright; botleft botright]
      V = inv(V_inv)

      term1 = (temp_lambdas[1] * tXy) + (Sigma_beta_inv * mu_beta)
      term2 = temp_lambdas[1] * tZy
      zeta = [term1; term2]

      Vchol=transpose(chol(V))
      temp_thetas = (V * zeta) + (Vchol * rand(Normal(0,1),p+q))
    end
      thetas[i,:] = temp_thetas'
      lambdas[i, :] = temp_lambdas
      sigmas[i, :] = temp_sigmas
  end

  return [thetas sigmas]
end


Y = readtable("nbadaty.csv")
X = readtable("nbadatx.csv")
Z = readtable("nbadatz.csv")
initialtheta = readtable("nbadatinit.csv")
DatX = convert(Array{Float64,2}, X)[:,2:end]
DatY = convert(Array{Float64,2}, Y)[:,2:end]
DatZ = convert(Array{Float64,2}, Z)[:,2:end]
thetastart = convert(Array{Float64,2}, initialtheta)[:,2]

iterations = 2
for l in 1:iterations
  @time dataoutput = GibbsLMMT(500000, 500000, 1, DatY, DatX, DatZ, [0.26872485; 0.07814486],
    eye(2) * 100, [1.0, 1.0], [1.0,1.0], 100, thetastart)
  describe(convert(DataFrame, dataoutput))
  # writedlm(string("LME_t-effects_",l,".txt"),dataoutput)
end
```

Listing C.10: Julia code

GibbsLMMT input description:

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `ModelMatrixZ`: A sparse matrix filled with 0's and 1's that associates each observation to a random effect
- `prior_mean_beta`: A numeric vector for the mean parameter of the normal distribution of β
- `prior_cov_beta`: A numeric matrix for the covariance matrix parameter of the normal distribution of β
- `prior_gamma_shape`: A numeric vector that provides the shape parameter for the prior distribution of τ_e and τ_u respectively
- `prior_gamma_rate`: A numeric vector that provides the rate parameter for the prior distribution of τ_e and τ_u respectively
- `df`: A numeric value for the degrees of freedom for the distribution of u
- `start_theta`: A concatenated numeric vector of initial values for MCMC of β and u

C.3.4 MATLAB

```matlab
function [thetas, sigmas] = GibbsLMMt_effects(iterations, burnin, nthin, Response,
    ModelMatrixX, ModelMatrixZ, prior_mean_beta, prior_cov_beta, prior_gamma_shape,
    prior_gamma_rate, df, start_theta)

X = ModelMatrixX;
Z = ModelMatrixZ;
y = Response;
Sigma_beta = prior_cov_beta;
Sigma_beta_inv = inv(Sigma_beta);
mu_beta = prior_mean_beta;
a = prior_gamma_shape;
b = prior_gamma_rate;
W = [X, Z];
N = length(y);
p = size(X, 2);
q = size(Z, 2);

tXX = X' * X;
tXZ = X' * Z;
tZX = Z' * X;
tZZ = Z' * Z;
tXy = X' * y;
tZy = Z' * y;

thetas = repmat(0.0, iterations, p+q);
lambdas = repmat(0.0, iterations, 2);
sigmas = repmat(0.0, iterations, 2);
temp_thetas = start_theta;
temp_lambdas = repmat(0.0,1,2);
temp_sigmas = repmat(0.0,1,2);
eta = repmat(0.0, q,1);
postrate_e = 0.0 ;
postshape_e = 0.0;

V_inv = repmat(0.0, p+q,p+q);
D_eta = eye(q);
      postshape_e = a(1) + N * 0.5;
      postshape_u = a(2) + q * 0.5;
for i = 1:burnin
     postrate_e = b(1) + (norm(y - W * temp_thetas)^2)/2;
      postrate_u = b(2) + (norm(D_eta^0.5 * temp_thetas((p+1):end))^2)/2;

      temp_lambdas(1) =gamrnd(postshape_e, 1/postrate_e,1);
      temp_lambdas(2) = gamrnd(postshape_u, 1.0/postrate_u);
      temp_sigmas(1) = 1.0/sqrt(temp_lambdas(1));
      temp_sigmas(2) = 1.0/sqrt(temp_lambdas(2));

      for l=1:q
            etarate = (df + temp_lmabdas(2) * temp_thetas(p+l)^2)*0.5
            eta(l) = gamrnd((df+1)*0.5, 1/etarate)
      end


      T_lambda = temp_lambdas(1) * tXX + Sigma_beta_inv;
      T_lambda_inv = inv(T_lambda);
      M_lambda = eye(N) - (temp_lambdas(1) * X * T_lambda_inv * X');
      Q_lambda_eta = (temp_lambdas(1) * Z' * M_lambda * Z) + temp_lambdas(2) * D_eta;

      topleft = T_lambda;
      topright = temp_lambdas(1) * tXZ;
      botleft = temp_lambdas(1) * tZX;
      botright = temp_lambdas(1) * tZZ + temp_lambdas(2) * D_eta;

      V_inv = [topleft, topright; botleft, botright];
     V = inv(V_inv);

      term1 = (temp_lambdas(1) * tXy) + (Sigma_beta_inv * mu_beta);
      term2 = temp_lambdas(1) * tZy;
      zeta = [term1; term2];

      Vchol=transpose(chol(V));
      temp_thetas = (V * zeta) + (Vchol * normrnd(0,1,p+q,1) );
end
for i = 1:iterations
    for nth = 1:nthin
      postrate_e = b(1) + (norm(y - W * temp_thetas)^2)/2;
      postrate_u = b(2) + (norm(D_eta^0.5 * temp_thetas((p+1):end))^2)/2;

      temp_lambdas(1) =gamrnd(postshape_e, 1/postrate_e,1);
      temp_lambdas(2) = gamrnd(postshape_u, 1.0/postrate_u);
      temp_sigmas(1) = 1.0/sqrt(temp_lambdas(1));
      temp_sigmas(2) = 1.0/sqrt(temp_lambdas(2));



      T_lambda = temp_lambdas(1) * tXX + Sigma_beta_inv;
      T_lambda_inv = inv(T_lambda);
      M_lambda = eye(N) - (temp_lambdas(1) * X * T_lambda_inv * X');
      Q_lambda_eta = (temp_lambdas(1) * Z' * M_lambda * Z) + temp_lambdas(2) * D_eta;

      topleft = T_lambda;
      topright = temp_lambdas(1) * tXZ;
      botleft = temp_lambdas(1) * tZX;
      botright = temp_lambdas(1) * tZZ + temp_lambdas(2) * D_eta;

      V_inv = [topleft, topright; botleft, botright];
      V = inv(V_inv);

      term1 = (temp_lambdas(1) * tXy) + (Sigma_beta_inv * mu_beta);
      term2 = temp_lambdas(1) * tZy;
      zeta = [term1; term2];

      Vchol=transpose(chol(V));
      temp_thetas = (V * zeta) + (Vchol * normrnd(0,1,p+q,1) );
    end
      thetas(i,:) = temp_thetas';
      lambdas(i, :) = temp_lambdas;
      sigmas(i, :) = temp_sigmas;
end

end
```

Listing C.11: MATLAB Code

`GibbsLMMt_effects` **input description:**

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `ModelMatrixZ`: A sparse matrix filled with 0's and 1's that associates each observation to a random effect
- `prior_mean_beta`: A numeric vector for the mean parameter of the normal distribution of β
- `prior_cov_beta`: A numeric matrix for the covariance matrix parameter of the normal distribution of β
- `prior_gamma_shape`: A numeric vector that provides the shape parameter for the prior distribution of τ_e and τ_u respectively
- `prior_gamma_rate`: A numeric vector that provides the rate parameter for the prior distribution of τ_e and τ_u respectively

- `df`: A numeric value for the degrees of freedom for the distribution of u
- `start_theta`: A concatenated numeric vector of initial values for MCMC of β and u

Appendix D

Probit Regression–Improper Prior

D.1 AC algorithm

D.1.1 R/Rcpp/JAGS Workflow

```
######################################
### Probit DA Regression Workflow ###
######################################

library(msm)
library(Rcpp)
library(RcppEigen)
library(inline)
library(truncnorm)
library(coda)
library(rjags)

source("ProbitRegressionDA_Source_2016-08-09.R")
###################
### Pima datset ###
###################

set.seed(999)
mydata<-read.table("cleanPIMA.txt", header=TRUE)
# Logistic Regression (Frequentist analysis)
fit.glucose <- glm( test ~ glucose, family=binomial(link="probit"), data = mydata)
# Summary of fit
summary(fit.glucose)
Y = mydata$test[-c(76, 183, 343, 350, 503)]
Ydat = as.numeric(Y == "positive")
Y = Ydat
X = model.matrix(fit.glucose)
tXX = t(X)%*%X
txxinv = solve(tXX)
netiterations = 10000
n = length(Y)
p = ncol(X)

BetaMCMC = matrix(NA, nrow = netiterations , ncol = p)
BetaMCMC[1, ] = c(1,1)
z = matrix(NA, n,1)

iteration = 10

##################################
#### Running the Gibbs Sampler ###
##################################


#Running in R
set.seed(999)
for(l in 1:iteration){

  start = Sys.time()
```

```r
  BetaMCMC = GibbsProbit(ModelMatrixX = X, Response = Y,
                            betainitial = c(1,1), iterations = 100,
                            burnin = 100, nthin = 1)

  Sys.time()-start
  print(paste("This is iteration: ", l))
  print(paste("This is iteration: ", l))
  print(paste("This is iteration: ", l))
  print(paste("This is iteration: ", l))

  print(summary(as.mcmc(BetaMCMC)))
 # write.csv(x = MCMC, file = paste("ProbitRegressionDA_",l,"_iterationR_2016-07-20.csv",sep
    =""))

}


#Rcpp
set.seed(999)

for(l in 1:iteration){
  start=Sys.time()
  dat=GibbsProbitcpp(100, 100, 1, Y, as.matrix(X), rep(0,ncol(X)))
  Sys.time()-start
  print(paste("This is iteration: ", l))
  print(paste("This is iteration: ", l))
  print(paste("This is iteration: ", l))
  print(paste("This is iteration: ", l))

  print(summary(as.mcmc(dat)))
  # write.csv(x = MCMC, file = paste("ProbitRegressionDA_",
  #                                  l,"_iterationRcpp_2016-07-20.csv",sep=""))

}

#JAGS
# Numbers do not match
set.seed(999)
for(l in 1:iteration){

  jagsfit <- jags.model(file = "ProbitRegressionImproper.jags",
                         data = list('Response' = Y,
                                     'ModelMatrixX' = X,
                                     'N' = length(Y),
                                     'P' = ncol(X),
                                     'var' = 100000000),
                         inits = list('beta'= rep(0, ncol(X))),
                         n.chains=1,
                         n.adapt=0
  )
  start.time <- Sys.time()
  update(jagsfit, 100) # Obtain first 100,000 (burnin draws)
  MCMC.out <- coda.samples(jagsfit,
                            var = c("beta"),
                            n.iter = 1000000,  # Obtain the main 100,000 draws
                            thin = 1)
  Sys.time()  - start.time

  print(paste("This is iteration: ", l))
  print(paste("This is iteration: ", l))
  print(paste("This is iteration: ", l))
  print(paste("This is iteration: ", l))

  print(summary(MCMC.out)) # Notice the iterations being used
  # write.csv(x = MCMC, file = paste("ProbitRegressionDA_",l
  #                                  ,"_iterationJAGS_2016-07-20.csv",sep=""))

}
```

Listing D.1: AC Algorithm R Work Flow

D.1.2 Julia Code

```julia
using Distributions, DataFrames

function DAProbitModel(iterations, burn_in, nthin, Response, ModelMatrixX, startbeta)
  n = size(ModelMatrixX, 1) #rows
  p = size(ModelMatrixX, 2) #columns
  BetaMCMC = fill(0.0, iterations, p) #Store MCMC
  TempBetaMCMC = startbeta
  z = fill(0.0, n, 1)
  txx = transpose(ModelMatrixX) * ModelMatrixX
  txxinverse = inv(txx)
  V = transpose(chol(txxinverse)) #Cholesky Decomposition

  for i in 1:burn_in
      for j in 1:n
        center = ModelMatrixX[j, :] * TempBetaMCMC
        if (Response[j] == 0)
            z[j] = rand(Truncated(Normal(center[1], 1), -Inf, 0.0))
          end
        if (Response[j] == 1)
            z[j] = rand(Truncated(Normal(center[1], 1), 0.0, Inf))
          end
       end #end of generating z's
       BetaHat = txxinverse * transpose(ModelMatrixX) * z
     TempBetaMCMC = BetaHat + (V * rand(Normal(),p))
  end


  for i in 1:iterations
    for k in 1:nthin
      for j in 1:n
        center = ModelMatrixX[j, :] * TempBetaMCMC
        if (Response[j] == 0)
            z[j] = rand(Truncated(Normal(center[1], 1), -Inf, 0.0))
          end
        if (Response[j] == 1)
            z[j] = rand(Truncated(Normal(center[1], 1), 0.0, Inf))
          end
       end #end of generating z's

       BetaHat = txxinverse * transpose(ModelMatrixX) * z
       TempBetaMCMC = BetaHat + (V * rand(Normal(),p))
    end #end of thinning
  BetaMCMC[i,:] = transpose(TempBetaMCMC)
  end #end of burn+ iterations for loop

  return BetaMCMC
end #end of function

Y = readtable("PIMAmatrixY.csv")
X = readtable("PIMAmatrixX.csv")
DatX = convert(Array{Float64,2}, X)[:, 2:end]
DatY = convert(Array{Float64,2}, Y)[:, 2:end]
qbbeta = fill(0.0, size(DatX, 2), 1)

iterations = 10

for(l in 1:iterations)
  @time dataoutput = DAProbitModel(500000, 500000, 1, DatY, DatX, qbbeta, )
    describe(convert(DataFrame,dataoutput))
    #writedlm(string("ProbitRegression_DA_PIMA_",l,".txt"), dataoutput )
end
```

Listing D.2: Julia Code

D.1.3 MATLAB code

```matlab
function [BetaMCMC] = ProbitDA(iterations, burnin, nthin, ModelMatrixX, Response, startbeta)
    n = length(Response);
    X = ModelMatrixX;
    y = Response;
    p = size(X,2);
    BetaMCMC = repmat(0.0, iterations, p);

    tempbeta = startbeta;

    z = repmat(0.0, n,1);
    znew = repmat(0.0, n, 1);
    tXX = X' * X;
    txxinv = inv(tXX);
    V = transpose(chol(txxinv));

    for i = 1:burnin
        for j = 1:n
        center = X(j,:) * tempbeta;
         pd = makedist('Normal', center, 1);
            if(y(j) == 0)
                z(j) = random(truncate(pd,-inf,0));
            end
            if(y(j) == 1)
                z(j) = random(truncate(pd,0, inf));
            end
        end

        betahat = txxinv * X' * z;
        tempbeta = betahat + (V * normrnd(0,1,p,1));
    end


    for i = 1:iterations
        for nth= 1:nthin
            for j = 1:n
                center = X(j,:) * tempbeta;
                pd = makedist('Normal', center, 1);
                if(y(j) == 0)
                    z(j) = random(truncate(pd,-inf,0));
                end
                if(y(j) == 1)
                    z(j) = random(truncate(pd,0, inf));
                end
            end


            betahat = txxinv * X' * z;
            tempbeta = betahat + (V * normrnd(0,1,p,1));
        end
        BetaMCMC(i,:) = tempbeta;
    end
end
```

Listing D.3: MATLAB

D.2 PX-DA Algorithm

D.2.1 Source code

```

##################################
############# R Code #############
##################################

GibbsProbitHaar = function(iterations, burnin, nthin, Response, ModelMatrixX, betainitial){
  X <- ModelMatrixX
  Y <- Response
  tXX <- t(X) %*% X
  txxinv <- solve(tXX)
  n <- length(Y)
  p <- ncol(X)
  V <- t(chol(txxinv))
  BetaMCMC <- matrix(NA, nrow = iterations, ncol = p)
  tempbeta <- betainitial
  z <- matrix(NA, n,1)
  znew <- matrix(NA, n,1)
  zPrime <- matrix(NA, n,1)

  for(k in 1:burnin){
    for(j in 1:n){
      center <- t(X[j,]) %*% tempbeta
      if(Y[j] == 0){
        z[j] <- rtruncnorm(1, a = -Inf, b = 0, mean = center, sd = 1)
      }
      if(Y[j] == 1){
        z[j] <- rtruncnorm(1, a = 0, b = Inf, mean = center, sd = 1)
      }
    }
    for(j in 1:n){
      znew[j] <- (z[j] - (X[j,]) %*% txxinv %*% t(X) %*% z)^2
    }
    Summation <- sum(znew)
    GSquared <- rgamma(1, (n/2), (1/2) * Summation)
    zPrime <- sqrt(GSquared) * z

    betahat <- txxinv %*% t(X) %*% zPrime
    tempbeta <- betahat + V %*% rnorm(p)
  }

  for(i in 1:iterations){
    for(k in 1:nthin){
      for(j in 1:n){
        center <- t(X[j,]) %*% tempbeta
        if(Y[j] == 0){
          z[j] <- rtruncnorm(1, a = -Inf, b = 0, mean = center, sd = 1)
        }
        if(Y[j] == 1){
          z[j] <- rtruncnorm(1, a = 0, b = Inf, mean = center, sd = 1)
        }
      }
      for(j in 1:n){
        znew[j] <- (z[j] - (X[j,]) %*% txxinv %*% t(X) %*% z)^2
      }
      Summation <- sum(znew)
      GSquared <- rgamma(1, (n/2), (1/2) * Summation)
      zPrime <- sqrt(GSquared) * z

      betahat <- txxinv %*% t(X) %*% zPrime
      tempbeta <- betahat + V %*% rnorm(p)
    }
    BetaMCMC[i,] <- tempbeta
  }
  return(BetaMCMC)
}

##################################
########### Rcpp Code ############
##################################

src<- '
using Eigen::Map;
using Eigen::MatrixXd;
using Eigen::VectorXd;
using Eigen::Vector2d;
using Rcpp::as;

typedef Eigen::Map<Eigen::MatrixXd> MapMatd;
typedef Eigen::Map<Eigen::VectorXd> MapVecd;


int MCMCiter = Rcpp::as<int>(iterations);
int burnin = Rcpp::as<int>(Burnin);
int n_thin = Rcpp::as<int>(nthin);

Rcpp::NumericMatrix Xc(ModelMatrixX);
Rcpp::NumericVector Yc(Response);
Rcpp::NumericVector BetaInitialc(BetaInitial);

const MapMatd X(Rcpp::as<MapMatd>(Xc));
const MapVecd Y(Rcpp::as<MapVecd>(Yc));
const MapVecd Betainitial(Rcpp::as<MapVecd>(BetaInitialc));

```

```
int n = X.rows();
int p = X.cols();

const MatrixXd tXX = X.transpose() * X;
const MatrixXd tXXinverse = tXX.inverse();

MatrixXd betaMCMC(MCMCiter, p);
MatrixXd V(p,p);
VectorXd tempbeta = Betainitial;
VectorXd Z(n);
VectorXd betahat(p);
VectorXd normals(p);
double center = 0.0;
double tnormmean = 0.7978846; // mean of standard normal truncated mean on (0,Inf)
double numerator = 0.0;
double denominator = 0.0;
double temp = 0.0;
double Znew = 0.0;
double sum = 0.0;
double gsquared = 0.0;

V = tXXinverse.llt().matrixL();

RNGScope scp;

Rcpp::Function rtnorm("rtnorm");
Rcpp::Function rnorm("rnorm");
Rcpp::Function dnorm("dnorm");
Rcpp::Function pnorm("pnorm");
Rcpp::Function print("print");

  for(int k = 0; k < burnin; k++){
    for(int j = 0; j < n; j++){
      center = X.row(j) * tempbeta;

      if(Y[j] == 0.0){
        Z[j] = as<double>(rtnorm(1, center, 1, R_NegInf, 0));
      }

      if(Y[j] == 1.0){
        Z[j] = as<double>(rtnorm(1,center,1,0, R_PosInf));
      }
    }

    for(int m = 0; m < n; m++){
      Znew = pow((Z[m] - (X.row(m) * tXXinverse *  X.transpose() * Z)) , 2);
      sum = sum + Znew;
    }
    gsquared = as<double>(Rcpp::rgamma(1, (n / 2.0), 1.0 / (0.5 *sum)));
    Z = sqrt(gsquared) * Z;
    betahat = tXXinverse * X.transpose() * Z;

    normals = Rcpp::as<MapVecd>(Rcpp::rnorm(p));
    tempbeta = betahat + V * normals;
    sum = 0.0;
  }

for(int i = 0; i < MCMCiter; i++){
  for(int k = 0; k < n_thin; k++){
    for(int j = 0; j < n; j++){

      center = X.row(j) * tempbeta;

      if(Y[j] == 0.0){
        Z[j] = as<double>(rtnorm(1, center, 1, R_NegInf, 0));
      }

      if(Y[j] == 1.0){
        Z[j] = as<double>(rtnorm(1,center,1, 0, R_PosInf));
      }
    }

    for(int m = 0; m < n; m++){
      Znew = pow((Z[m] - (X.row(m) * tXXinverse *  X.transpose() * Z)) , 2);
      sum = sum + Znew;
    }
    gsquared = as<double>(Rcpp :: rgamma(1, (n / 2.0), 1.0 / (0.5 *sum)));
    Z = sqrt(gsquared) * Z;
    betahat = tXXinverse * X.transpose() * Z;
    normals = Rcpp :: as<MapVecd>(Rcpp :: rnorm(p));
    tempbeta = betahat + V * normals;
    sum = 0.0;
  }
betaMCMC.row(i) = tempbeta.transpose();
}

return Rcpp :: DataFrame :: create(Rcpp :: Named("Beta") = betaMCMC);
'
GibbsProbitHaarcpp = cxxfunction(signature(iterations = "int",
                                           Burnin = "int", nthin = "int",
                                           Response = "numeric",
                                           ModelMatrixX = "numeric",
                                           BetaInitial = "numeric"), src,
                                 plugin="RcppEigen")
```

Listing D.4: PX-DA Algorithm Source Code

GibbsProbitHaar input description:

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `betainitial`: A numeric vector of initial values for MCMC of β

GibbsProbitHaarcpp input description:

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `Betainitial`: A numeric vector of initial values for MCMC of β

D.2.2 R/Rcpp/JAGS Workflow

```
#set directory
getwd()
setwd()
getwd()
setwd()


#call libraires and source
library(msm)
library(Rcpp)
library(RcppEigen)
```

```r
library(coda)
library(inline)
library(rjags)
library(truncnorm)

set.seed(999)
mydata<-read.table("cleanPIMA.txt", header=TRUE)
# Logistic Regression (Frequentist analysis)
fit.glucose <- glm( test ~ glucose, family=binomial(link="probit"), data = mydata)
# Summary of fit
summary(fit.glucose)
Y = mydata$test[-c(76, 183, 343, 350, 503)]
Ydat = as.numeric(Y == "positive")
Y = Ydat
X = model.matrix(fit.glucose)
tXX = t(X)%*%X
txxinv = solve(tXX)
iterations = 10000
n = length(Y)
p = ncol(X)

iteration = 4


###################################
############# R Code #############
###################################


set.seed(999)
for (l in 1:iteration){
start = Sys.time()
Beta = GibbsProbitHaar(ModelMatrixX = X, Response = Y,
                       betainitial = c(1,1), iterations = 100,
                       burnin = 100, nthin = 1)

Sys.time()-start

print(paste("############################# "))
print(paste("This is iteration: ", l))
print(paste("############################# "))

print(summary(Beta))

#write.csv(x = MCMC, file = paste("ProbitRegressionPXDA_",l,"_iterationR_2016-07-20.csv",sep
    =""))
}


###################################
########### Rcpp Code ############
###################################

set.seed(999)
for (l in 1:iteration) {
start = Sys.time()
dat = GibbsProbitHaarcpp(100, 100, 1, Y, X, rep(0,ncol(X)))
Sys.time()-start

print(paste("############################# "))
print(paste("This is iteration: ", l))
print(paste("############################# "))

print(summary(as.mcmc(dat)))

#  write.csv(x = MCMC,
#            file = paste("ProbitRegressionPXDA_",l
#                         ,"_iterationRcpp_2016-07-20.csv",sep=""))

}
```

Listing D.5: PX-DA Algorithm Work Flow

D.2.3 Julia Code

```julia
using Distributions, DataFrames

function PXDAProbitModel(iterations, burn_in, nthin, Response, ModelMatrixX, startbeta)
  n = size(ModelMatrixX, 1) #rows
  p = size(ModelMatrixX, 2) #columns
  BetaMCMC = fill(0.0, iterations, p) #Store MCMC
  TempBetaMCMC = startbeta
  z = fill(0.0, n, 1)
  znew = fill(0.0, n, 1)
  txx = transpose(ModelMatrixX) * MatrixX
  txxinverse = inv(txx)
  V = transpose(chol(txxinverse)) #Cholesky Decomposition

  for i in 1:burn_in
      for j in 1:n
        center = ModelMatrixX[j, :] * TempBetaMCMC
        if (Response[j] == 0)
            z[j] = rand(Truncated(Normal(center[1], 1), -Inf, 0.0))
          end
        if (Response[j] == 1)
            z[j] = rand(Truncated(Normal(center[1], 1), 0.0, Inf))
          end
       end #end of generating z's
       for m in 1:n
        znew[m] = ((z[m] - (ModelMatrixX[m,:] * txxinverse * transpose(ModelMatrixX) * z))[1])
    ^2
      end
      Summation = sum(znew)
      GSquare = rand(Gamma((n/2.0) , (1/((1.0/2.0) * Summation))))
      zPrime = sqrt(GSquare) * z
      BetaHat = txxinverse * transpose(ModelMatrixX) * zPrime
      TempBetaMCMC = BetaHat + (V * rand(Normal(),p))
  end


  for i in 1:iterations
    for k in 1:(nthin)
      for j in 1:n
        center = ModelMatrixX[j, :] * TempBetaMCMC
        if (Response[j] == 0)
            z[j] = rand(Truncated(Normal(center[1], 1), -Inf, 0.0))
          end
        if (Response[j] == 1)
            z[j] = rand(Truncated(Normal(center[1], 1), 0.0, Inf))
          end
       end #end of generating z's
       for m in 1:n
        znew[m] = ((z[m] - (ModelMatrixX[m,:] * txxinverse * transpose(ModelMatrixX) * z))[1])
    ^2
      end
      Summation = sum(znew)
      GSquare = rand(Gamma((n/2.0) , (1/((1.0/2.0) * Summation))))
      zPrime = sqrt(GSquare) * z
       BetaHat = txxinverse * transpose(ModelMatrixX) * zPrime
       TempBetaMCMC = BetaHat + (V * rand(Normal(),p))
    end #end of thinning
  BetaMCMC[i, :] = transpose(TempBetaMCMC)
  end #end of burn+ iterations for loop


  return BetaMCMC[burn_in:end, :]
end #end of function

Y = readtable("PIMAmatrixY.csv")
X = readtable("PIMAmatrixX.csv")
DatX = convert(Array{Float64,2}, X)[:, 2:end]
DatY = convert(Array{Float64,2}, Y)[:, 2:end]
qbbeta = fill(0.0, size(DatX, 2), 1)

iterations = 1

```

```
for(l in 1:iterations)
  @time dataoutput = PXDAProbitModel(500000, 500000, 1, DatY, DatX, qbbeta)
    describe(convert(DataFrame,dataoutput))
   # writedlm(string("ProbitRegression_PXDA_PIMA_",l,".txt"), dataoutput )
end
```

Listing D.6: Julia Code

DAProbitModel input description

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `startbeta`: A numeric vector of initial values for MCMC of β

D.2.4 MATLAB

```
function [BetaMCMC] = ProbitPXDA(iterations, burnin, nthin, ModelMatrixX, Response, startbeta)
    n = length(Response);
    X = ModelMatrixX;
    y = Response;
    p = size(X,2);
    BetaMCMC = repmat(0.0, iterations, p);
    var=1
    tempbeta = startbeta;

    z = repmat(0.0, n,1);
    znew = repmat(0.0, n, 1);
    tXX = X' * X;
    txxinv = inv(tXX);
    gshape = n/2
    for i = 1:burnin
        for j = 1:n
        center = X(j,:) * tempbeta;
            if(y(j) == 0)
                z(j) = random(truncate(makedist('Normal', center, var),-inf,0));
            end
            if(y(j) == 1)
                z(j) = random(truncate(makedist('Normal', center, var),0, inf));
            end
        end

        for m =1:n
            znew(m) = (z(m) - (X(m,:) * txxinv * X' * z) )^2;
        end
        summation = sum(znew);
        gsq = gamrnd(gshape, 1/ (0.5*summation));
        zprime = sqrt(gsq) * z;
        betahat = txxinv * X' * zprime;
       V = transpose(chol(txxinv));
        tempbeta = betahat + (V * normrnd(0,1,p,1));
    end


    for i = 1:iterations
        for nth= 1:nthin
            for j = 1:n
                center = X(j,:) * tempbeta;

                if(y(j) == 0)
                    z(j) = random(truncate(makedist('Normal', center, var),-inf,0));
                end
                if(y(j) == 1)
                    z(j) = random(truncate(makedist('Normal', center, var),0, inf));
                end
            end

            for m =1:n
                znew(m) = (z(m) - (X(m,:) * txxinv * X' * z) )^2;
            end
            summation = sum(znew);
            gsq = gamrnd(gshape, 1/ (0.5*summation));
            zprime = sqrt(gsq) * z;
            betahat = txxinv * X' * zprime;
            V = transpose(chol(txxinv));
            tempbeta = betahat + (V * normrnd(0,1,p,1));
        end
        BetaMCMC(i,:) = tempbeta;
    end
end
```

Listing D.7: MATLAB

PXDAProbitModel input description

- `iterations`: Net length of MCMC chain for main sample
- `burnin`: Number of draws for MCMC chain to initialize before main sample
- `nthin`: Number of draws to consider before storing main sample, i.e. every second; every third; etc.
- `Response`: A numeric vector of observed data for linear model
- `ModelMatrixX`: A numeric matrix of predictors for linear model
- `startbeta`: A numeric vector of initial values for MCMC of β

Bibliography

ALBERT, J. H. and CHIB, S. (1993). Bayesian analysis of binary and polychotomous response data. *Journal of the American Statistical Association*, **88** 669–679.

CHRISTENSEN, R., JOHNSON, W., BRANSCUM, A. and HANSON, T. (2010). *Bayesian Ideas and Data Analysis: An Introduction for Scientists and Statisticians*. CRC Press.

DIACONIS, P., KHARE, K. and SALOFF-COSTE, L. (2008). Gibbs sampling, exponential families and orthogonal polynomials (with discussion). *Statistical Science*, **23** 151–200.

HARVILLE, D. A. (1997). *Matrix Algebra From a Statistician's Perspective*. Springer.

JONES, G. L. and HOBERT, J. P. (2001). Honest exploration of intractable probability distributions via Markov chain Monte Carlo. *Statistical Science*, **16** 312–34.

LATUSZYŃSKI, K., MIASOJEDOW, B. and NIEMIRO, W. (2013). Nonasymptotic bounds on the estimation error of MCMC algorithms. *Bernoulli* 2033–2066.

LIU, J. S. and WU, Y. N. (1999). Parameter expansion for data augmentation. *Journal of the American Statistical Association*, **94** 1264–1274.

ROBERTS, G. O. and ROSENTHAL, J. S. (2001). Markov chains and de-initializing processes. *Scandinavian Journal of Statistics*, **28** 489–504.

ROMÁN, J. C. (2012). *Convergence Analysis of Block Gibbs Samplers for Bayesian General Linear Mixed Models*. Ph.D. thesis, Department of Statistics, University of Florida.

ROMÁN, J. C. and HOBERT, J. P. (2012). Convergence analysis of the Gibbs sampler for Bayesian general linear mixed models with improper priors. *Annals of Statistics*, **40** 2823–2849.

ROMÁN, J. C. and HOBERT, J. P. (2015). Geometric ergodicity of Gibbs samplers for Bayesian general linear mixed models with proper priors. *Linear Algebra and its Applications*, **473** 54–77.

ROMÁN, J. C., JACOBS, N. H. and DE LA CRUZ, R. (2016). Geometric ergodicity of gibbs samplers for bayesian general linear mixed models with t-distributed effects. Tech. rep., San Diego State University.

ROSENTHAL, J. S. (1995). Minorization Conditions and Convergence rates for Markov Chain Monte Carlo. *Journal of the American Statistical Association*, **90** 558–566.